Towards Larger Scale Collective Operations in the Message Passing Interface

Martin Rüfenacht

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2020

Abstract

Supercomputers continue to expand both in size and complexity as we reach the beginning of the exascale era. Networks have evolved, from simple mechanisms which transport data to subsystems of computers which fulfil a significant fraction of the workload that computers are tasked with. Inevitably with this change, assumptions which were made at the beginning of the last major shift in computing are becoming outdated.

We introduce a new latency-bandwidth model which captures the characteristics of sending multiple small messages in quick succession on modern networks. Contrary to other models representing the same effects, the pipelining latency-bandwidth model is simple and physically based. In addition, we develop a discrete-event simulation, Fennel, to capture non-analytical effects of communication within models.

AllReduce operations with small messages are common throughout supercomputing, particularly for iterative methods. The performance of network operations are crucial to the overall time-to-solution of an application as a whole. The Message Passing Interface standard was introduced to abstract complex communications from application level development. The underlying algorithms used for the implementation to achieve the specified behaviour, such as the recursive doubling algorithm for AllReduce, have to evolve with the computers on which they are used.

We introduce the recursive multiplying algorithm as a generalisation of recursive doubling. By utilising the pipelining nature of modern networks, we lower the latency of AllReduce operations and enable greater choice of schedule. A heuristic is used to quickly generate a near-optimal schedule, by using the pipelining latency-bandwidth model.

Alongside recursive multiplying, the endpoints of collective operations must be able to handle larger numbers of incoming messages. Typically this is done by duplicating receive queues for remote peers, but this requires a linear amount of memory space for the size of the application. We introduce a single-consumer multiple-producer queue which is designed to be used with MPI as a protocol to insert messages remotely, with minimal contention for shared receive queues.

Lay Summary

My research asks whether any approaches exist to enable larger scale communication within modern supercomputers, which are built out of large numbers of smaller computers connected with a network. While current algorithms are suitable for the size of supercomputers designed two decades ago, newer methods need to be developed to take advantage of modern networks.

I develop a simple mathematical representation of the behaviour for small messages on these modern networks, which is used to create a new algorithm to efficiently compute sums of numbers spread across an entire supercomputer. The key feature of the modern networks which we exploit is that sending two messages, one after another, requires less time than sending two single messages. We also develop a new algorithm to reduce congestion, allowing for many peers to communicate with less memory usage.

I show the new summation algorithm is faster than previous algorithms for small messages, which allows large simulations to complete in less time. This helps all sciences which use simulation as a core component of their work, ranging across a wide variety of topics including weather forecasting, inexpensive safety testing, and designing efficient aircraft.

Acknowledgements

First, I want to thank Dr Stephen Booth. He provided guidance and understanding when needed and always enthusiastically discussed ideas with me. Providing freedom to explore my curiosity has been an invaluable gift. To Dr Mark Bull, thank you for everything. From the methodology to the proof-reading this thesis would not have come together if it were not for your efforts and support. I am grateful to Dr James Cheney and Dr Daniel Holmes for always providing outside counsel and encouragement to continue with my research.

I am fortunate to have been a member of the Centre for Doctoral Training in Pervasive Parallelism, especially with Professor Murray Cole involved in its organisation. Another important thanks is to Professor Anthony Skjellum who gave me the chance to explore research outside the PhD, which reinvigorated my interest in it. A special thanks belongs to my parents, Claudine and Peter Rüfenacht for always supporting me throughout my time at the University of Edinburgh. Finally, I want to thank all my friends for always being willing for a coffee, a distraction or a board game when it was needed or not.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Martin Rüfenacht*)

Table of Contents

List of Figures **xi**

List of Tables **xv**

Acronyms **xvii**

1 Introduction **1**

1.1 Structure 3

1.2 Publications 4

2 History of High Performance Computing **5**

2.1 Introduction 5

2.2 Beginning of Computing 6

2.3 Early High Performance Computing 7

2.4 Recent History of Supercomputing 9

2.4.1 Message Passing Interface 12

2.4.2 PGAS Interfaces 14

2.5 Today 15

3 Experimental Setup **19**

3.1 Introduction 19

3.2 Hardware 20

3.2.1 Aries Network Interface Controller 21

3.3 Software 23

3.3.1 Cray Shared Memory 23

3.3.2 Cray Distributed Memory Applications 24

3.3.3 Cray user-level Generic Network Interface 26

3.3.4 Cray MPI 26

4 Performance Modelling **29**
4.1 Introduction . . . 30
4.1.1 Contributions . . . 30
4.1.2 Overview . . . 31
4.2 Prior Performance Models . . . 31
4.2.1 Bulk Synchronous Parallel . . . 31
4.2.2 Latency-Bandwidth Model Family . . . 33
4.2.3 LogP Model Family . . . 34
4.2.4 LoP . . . 35
4.2.5 LogfP . . . 36
4.3 Pipelining Latency-Bandwidth Model . . . 36
4.3.1 Validation . . . 42
4.3.2 Larger Message Sizes . . . 42
4.3.3 Model Comparison . . . 44
4.4 Fennel Model Simulator . . . 48
4.4.1 Background . . . 49
4.4.1.1 ROSS . . . 50
4.4.1.2 Parsim . . . 50
4.4.1.3 SST . . . 51
4.4.1.4 LogGOPSim . . . 51
4.4.2 Program Representation . . . 52
4.4.3 Simulator Architecture . . . 54
4.4.4 Capabilities . . . 55
4.4.5 Validation . . . 58

5 Recursive Multiplying **59**
5.1 Introduction . . . 60
5.1.1 Overview . . . 60
5.1.2 Contributions . . . 60
5.2 Background . . . 61
5.2.1 MPI_Allreduce Definition . . . 61
5.2.2 Algorithms . . . 62
5.2.2.1 Fan-In/Fan-Out . . . 63
5.2.2.2 Recursive Doubling . . . 63
5.2.2.3 Composite Algorithms . . . 64

5.2.2.4 Elimination 67
5.3 Related work 68
5.4 Recursive Multiplying Algorithm 69
5.4.1 Derivation 69
5.4.2 Values Outside The Domain 76
5.4.3 Implementation 80
5.4.4 Heuristic Schedules 84
5.4.5 Large Messages 87
5.5 Experimental Results 89
5.5.1 Environment 89
5.5.2 AllReduce Benchmark 90
5.5.3 AllReduce Schedule Comparison 90
5.5.4 Message Size Scalability 94
5.5.5 AllReduce Model Comparison 96
5.5.6 Block-size Systematic Error Analysis 98
5.5.7 Experimental-Model Correlation 98
5.5.8 Cray MPI Comparison 100
5.6 Simulator Exploration 100
5.6.1 Factored Schedules 100
5.6.2 Splitting & Merging 103
5.6.3 3-2 & 2-1 Elimination 105

6 Two-sided MPI Receive Queue **109**
6.1 Introduction 110
6.1.1 Contributions 111
6.1.2 Overview 111
6.2 Prior Work 111
6.2.1 Cray MPI 111
6.2.2 EMPI4Re 113
6.2.3 Related Work 115
6.3 SCMP Algorithm 115
6.4 Experiments 119
6.4.1 Environment 119
6.4.2 Latency 120
6.4.3 Memory Scaling 121

6.4.4 Temporal Scaling 124

7 Conclusion **127**

7.1 Summary 127

7.2 Further Work 129

7.2.1 Performance Modelling 129

7.2.2 Recursive Multiplying 129

7.2.3 Receive Queue Mechanism 130

Bibliography **131**

List of Figures

1.1 Increasing Node Count 2

2.1 Human Computers 6
2.2 Flynn's Taxonomy 8
2.3 4D Hypercube Topology 10
2.4 Butterfly Topology 10
2.5 3D Torus Topology 11
2.6 Comparing Point-to-Point And Collective Communications 14
2.7 Fat-Tree Topology 17

3.1 Cray XC30 Blade Structure 20
3.2 Cray XC30 Rank-1 and Rank-2 Topology 21
3.3 Cray XC30 Rank-3 Topology 22
3.4 Cray XC30 Software Stack 24
3.5 MPICH2 Layer Diagram 27

4.1 BSP Model 32
4.2 LogP Model 34
4.3 LoP Model 36
4.4 LogfP Model 37
4.5 Pipelining Latency-Bandwidth Model 37
4.6 Pipelining Latency-Bandwidth Data 39
4.7 Pipelining Latency-Bandwidth Regression 40
4.8 Message Size Pipelining Capability 43
4.9 LogfP Round Trip Time 46
4.10 ARCHER Round Trip Time 46
4.11 *tds* Round Trip Time 47
4.12 PingPong Fennel Representation 52

4.13 Recursive Doubling Program Representation 53
4.14 Fennel Simulator UML 56
4.15 Fennel PingPong Simulation 57
4.16 Fennel Recursive Doubling Simulation 57

5.1 MPI_Allreduce Operation 61
5.2 Binomial Tree AllReduce 62
5.3 Recursive Doubling AllReduce 64
5.4 Composite AllReduce 66
5.5 Binary Block AllReduce 68
5.6 Recursive Multiplying Surface Plot 70
5.7 Machine Parameter Ratio Comparison 71
5.8 b_{opt} Function Plot 74
5.9 b_{upper} Function Plot 75
5.10 AllReduce Using Recursive Multiplying 75
5.11 Collapse/Expand Method Illustration 77
5.12 Hypercuboid Illustration of AllReduce 78
5.13 Hypercuboid View of Prime Merging 78
5.14 Hypercuboid View of Prime Merging Multiple Processes 79
5.15 Schedule Count Plot 85
5.16 Heuristic Efficiency Plot 88
5.17 Experimental Comparison of Recursive Doubling and Recursive Multiplying 92
5.18 Experimental Message Size Schedule Comparison for 8 processes. . . 95
5.19 Experimental Message Size Schedule Comparison for 64 Processes . 95
5.20 Recursive Multiplying Model Comparison 96
5.21 Experimental Block-size Analysis 99
5.22 Correlation Plot of Model and Experimental Minimums 101
5.23 Correlation Plot of Model and Experimental Medians 101
5.24 Experimental MPI Overhead Comparison 102
5.25 Fennel Program Representation 103
5.26 Simulation of Recursive Doubling for Power of Two Processes 104
5.27 Simulation of Recursive Multiplying 104
5.28 Simulation of Collapse/Expand Recursive Doubling 105
5.29 Simulation of Prime Merging Showing Minimal Skew 106

5.30 Simulation of Prime Merging with Skew 106
5.31 Simulation of 3-2 Elimination 107

6.1 SMSG Queue Mechanism 112
6.2 Remote Lock Protocol Queue Mechanism 113
6.3 Lock Bit-wise Encoding 114
6.4 SCMP Queue Mechanism 116
6.5 SCMP Bit-wise Encoding 117
6.6 Experimental Queue Mechanism Latency Comparison 120
6.7 Virtual Memory Usage by Queue Mechanism 122
6.8 Total Memory Usage Illustration 123
6.9 Locked Queue Mechanism Insertion Time 125
6.10 SCMP Queue Mechanism Insertion Time 125
6.11 *smsg* Queue Mechanism Insertion Time 126
6.12 *msgq* Queue Mechanism Insertion Time 126

List of Tables

4.1 Model Data Fit by Message Size 44

5.1 Possible Factorisations of Recursive Multiplying 76
5.2 Heuristically Generated Schedules 87
5.3 Recursive Doubling and Recursive Multiplying Comparison 91

Acronyms

AMO Atomic Memory Operation.

APGAS Asynchronous Partitioned Global Address Space.

API Application Programming Interface.

BSP Bulk Synchronous Parallel.

BTE Block Transfer Engine.

CPU Central Processing Unit.

DAG Directed Acyclic Graph.

ENIAC Electronic Numerical Integrator and Computer.

EPCC Edinburgh Parallel Computing Centre.

FFTW Fastest Fourier Transform in the West.

FMA Fast Memory Access.

GASPI Global Address Space Programming Interface.

GPU Graphics Processing Unit.

HPC High Performance Computing.

IBM International Business Machines Corporation.

IOMMU Input Output Memory Management Unit.

MIMD Multiple-Instruction Multiple-Data.

MPI Message Passing Interface.

MPI Message Passing Interface.

NIC Network Interface Controller.

PE Processing Element.

PGAS Partitioned Global Address Space.

PMI Process Management Interface.

PRAM Parallel Random Access Machine.

RAM Random Access Machine.

RDMA Remote Direct Memory Access.

RMA Remote Memory Access.

SOC System On a Chip.

SPMD Single-Program Multiple-Data.

TLB Translation Look-aside Buffer.

uGNI user-level Generic Network Interface.

Chapter 1

Introduction

Supercomputers in use today are primarily used for scientific simulations which require tremendous amounts of computational power. These machines are more difficult to program correctly, compared to desktop computers, due to the inherent parallelism built into the architecture. Machines today are built for software to use multiple Graphics Processing Units (GPUs) and multiple Central Processing Units (CPUs) per node. In addition, node parallelism is present, using multiple nodes to achieve a solution cooperatively. Node counts have increased by approximately two orders of magnitude since distributed memory supercomputers were first introduced as seen on Figure 1.1.

By abstracting various tasks required through libraries such as Message Passing Interface (MPI) it is possible to simplify the work done by application developers for inter-node communication. By using MPI the application developers need to only know the interface given, without knowing the underlying complexity of the networking algorithms. This is particularly true for collective operations provided by MPI.

Collective operations are used throughout scientific computing due to their intrinsic mapping to the concept of global data transformations. The most used collective operation is AllReduce[65, 71, 94, 105]. AllReduce is a reduction operation, such as a summation, which also places the result into the memory of all participating processes. This thesis works towards addressing the scalability and latency of AllReduce for larger scale supercomputers.

Scalability is an important aspect of an algorithm, because it determines how efficiently an operation is performed. When strong scaling is applied, keeping the problem size constant while increasing the computational resources, the scaling behaviour of collectives dominants the computation. Most operations can be performed in a time with a lower limit of $O(log_2 N)$, with N being the number of processes. Therefore

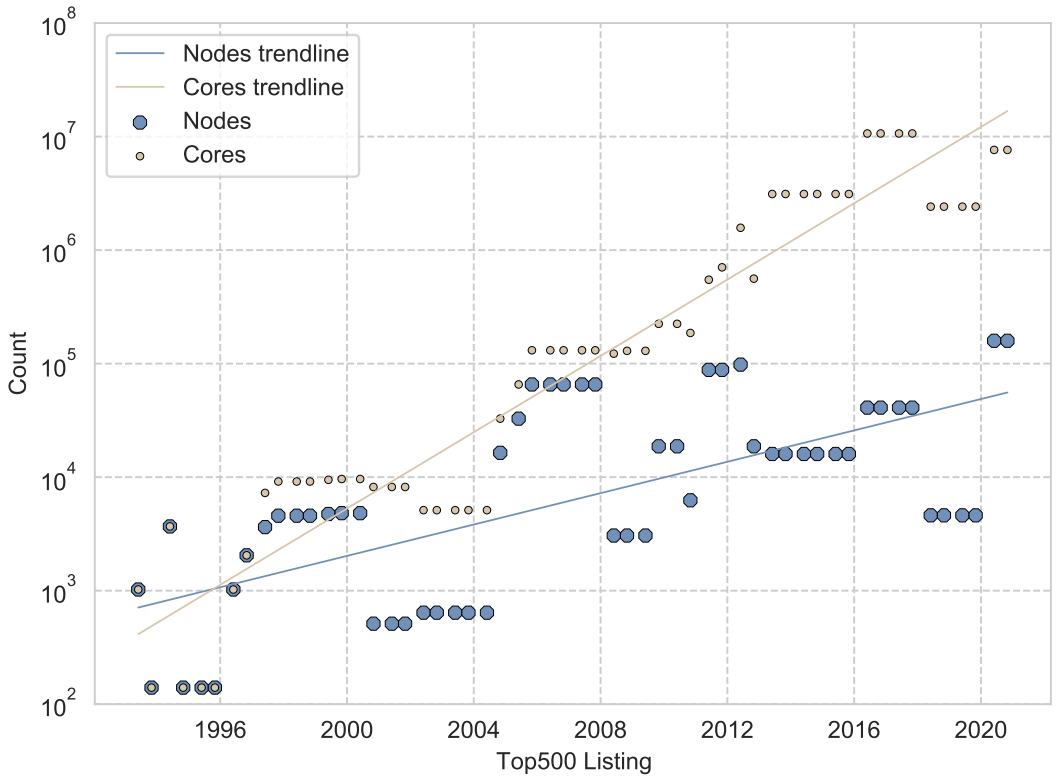


Figure 1.1: This plot shows node and core counts for all machines from the Top500[30] top ranked machine from June 1993 to November 2020.

optimizing collective operations, such as AllReduce, is paramount.

The parallel efficiency often tends to reduce with strong scaling. With weak scaling, scaling the problem size to the computational resources, collectives are less important since the computational requirements are also increased. Weak scaling attempts to keep the parallel efficiency constant, but the communication cost grows logarithmically which dominates the calculation cost while keeping the computational cost constant.

1.1 Structure

This section provides an overview of the thesis by giving the reader a short description of what is discussed within each chapter.

Chapter 2 describes some historical aspects of High Performance Computing (HPC) and the architectures of today's supercomputers. The chapter finishes with the current status of HPC and illustrates how the field has arrived at this point.

Chapter 3 gives a brief overview of the Cray XC30 supercomputer, ARCHER, which is used throughout this thesis for experimental and theoretical work. It discusses the capabilities of, and interfaces present on, all modern supercomputers.

Chapter 4 introduces our novel model, the pipelining latency-bandwidth model, suited for modelling of small-message operations on ARCHER. In addition, it introduces the *Fennel* simulator to analyse algorithms and models in a more fine-grained manner than is possible through a purely analytical approach.

Chapter 5 introduces the *recursive multiplying* algorithm for AllReduce operations based on the pipelining latency-bandwidth model. An extensive analytical analysis is given to explore the *recursive multiplying* algorithm. Experimental results are presented, illustrating the improvement in performance compared to prior methods. Finally, a heuristic method is introduced to determine a suitable higher performance schedule quickly.

Chapter 6 introduces a remote queue algorithm for a memory limited environment for the point-to-point operations internal to an MPI library. Experimental results are given to show that the scalability of the method is similar to situations where memory is not limited.

Chapter 7 summarises the thesis and discusses various future paths forward for many of the topics of the thesis. Primarily, extensions to the *recursive multiplying* algorithm are discussed due to its vast potential to extend to other collective operations.

1.2 Publications

During the research of this thesis the following publications were published in various conferences and journals:

Generalisation of Recursive Doubling for AllReduce[98]. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI16)*. Martin Ruefenacht, Mark Bull, Stephen Booth.

Generalisation of Recursive Doubling for AllReduce: Now with simulation[99]. In *Parallel Computing*. Martin Ruefenacht, Mark Bull, Stephen Booth.

A Large-Scale Study of MPI Usage in Open-Source HPC Applications[71]. In *Proceedings of the International Conference for High Performance Computing (SC19)*. Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, Nawrin Sultana.

Understanding the use of MPI in Exascale Proxy Applications[105]. In *Concurrency and Computation: Practice and Experience*. Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, Kathryn Mohror.

Chapter 2

History of High Performance Computing

2.1 Introduction

This chapter intends to guide the reader through the history of computing relevant to HPC. In doing so we explore the applications of the computers from the around the Second World War to the current era, in which scientific computing is the driving force behind development for HPC. Throughout the history of HPC software has been written to optimize for the underlying hardware at the time. We show the progression of hardware and software concepts as a linear evolution through this time.

To understand the history of the architectures and reasons for the designs, we need to understand the design decisions which have been factored into the development of computers from the beginning. The current state of architectures is one of many intermediate points in design where history could have progressed to. Understanding the different paths that could have been taken is important in understanding the theoretical models for supercomputers and potential future directions.

The goal of this chapter is to understand why current supercomputers, as of the Cray XC30 from 2012, are designed the way they are. We discuss several early supercomputers which have influenced the design of modern supercomputers.

Section 2.2 will introduce the earliest era of HPC and the need for it. Section 2.3 will show the progression of HPC towards the architectures of today. Section 2.4 discusses the introduction of the hardware and software which is present in modern supercomputing. Section 2.5 illustrates some of the current trends of hardware and software.

Figure 2.1: Women employed by NASA as human computers responsible for calculating launch windows, fuel consumption, and trajectories.[88]

2.2 Beginning of Computing

The advent of computing on a large scale occurred during the Second World War due to the need for tasks which could not be done analytically[66]. This was the first time that mathematics in this form was required, because of the need for cryptography and the breaking of the given cryptographic algorithms of rivals. Many attempts at the Enigma machine were attempted which were analytical, and while these attacks yielded success the difficulty of decryption on a large scale was still present with many cipher keys being used.

Until the Second World War computers were humans who performed large sets of mathematics in order to calculate a specific function, Figure 2.1 shows this at NASA. This became a bottleneck, because of the sheer number of calculations that any specific person would have to perform. A late example of this was the early space projects by the United States of America in which human computers were used to analyse test data and calculate trajectories.

When cryptography became popular prior to the Second World War, human computers were the only way to compute large scale mathematical problems. The use of humans for this task was, however, not very scalable and the need for a reliable and faster machine was apparent[40].

The first task specific machines were the *Bomba*[119] machines developed by Marian Regewski, a Polish cryptologist. The design of these machines were later given to the British Government Code and Cypher School to be used to decrypt German radio traffic at Bletchley Park[55]. The Manhattan Project, the development of the first nuclear weapon, also required computation done by human computers.

The concept of software as it is known today did not exist for these earliest machines. The machines were purpose built and served only that specific function. The hardware as designed did have configurable components to adjust to the requirements, but these were only similar to program parameters. The modern instruction set and instruction data functionality were not present in old machines. If the functionality had to be changed the entire machine had to be rebuilt.

Among the first general purpose computers was Electronic Numerical Integrator and Computer (ENIAC), commissioned at the end of the Second World War[111]. It was used for artillery trajectory calculations and weather simulations. The ENIAC computer was initially similar to the older machines which were purpose built, but in 1947 it was re-engineered to allow for program storage. This enabled it as one of the first general purpose machines to be programmed with the modern form of software, which is stored in memory alongside program data. The first program run on ENIAC was a simulation of atomic fission by the Los Alamos National Laboratory[79].

From the time when ENIAC was developed many larger general-purpose computers were developed throughout the 1940s and 1950s. All general-purpose computers developed until 1957 were only capable of being programmed with low-level assembly software. International Business Machines Corporation (IBM) released the first version of Fortran[64] with the IBM 704 in 1957. This was the introduction of the first high-level language in which concepts did not have to be expressed in a machine-readable format. This lead the creation of many programming languages and much of computer science as it is known today.

2.3 Early High Performance Computing

After the Second World War computers, both analog and digital, had been established as the effective computational resources instead of human computers. These early computers were typically large, of the size of rooms or warehouses, and were mostly used by universities or national laboratories.

The replacement of vacuum tubes with transistors in the 1950s yielded smaller and

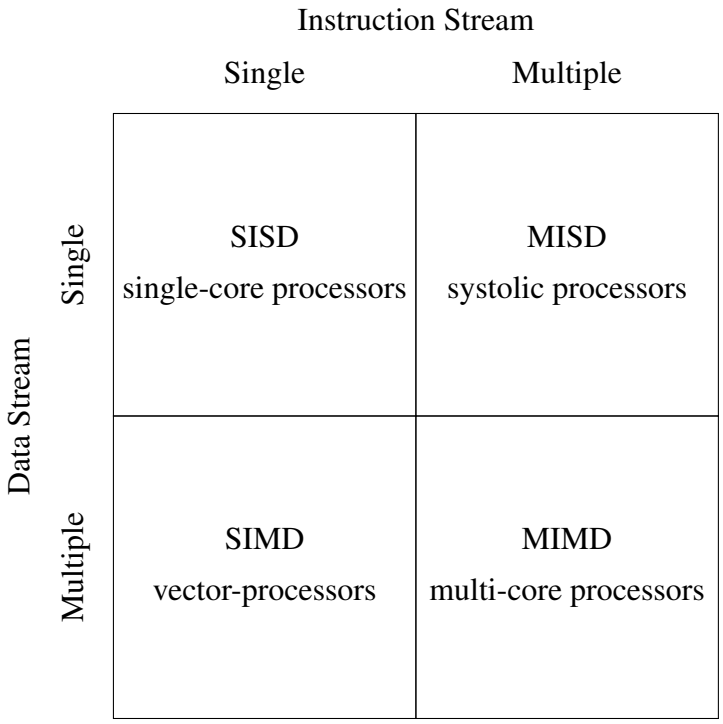


Figure 2.2: Illustration of Flynn's Taxonomy with examples for each class.

therefore more powerful computers. In the late 1950s the first MOSFET was invented at Bell Telephone Laboratories. This resulted in high density integrated circuits being used for computer construction from then onward. In 1965 Gordon Moore postulated the growth rate of integrated circuits, now known are Moore's Law[83]. This density increase was the main driving force behind performance increases for several decades alongside Dennard scaling[28].

Computers up until this point operated in a single-instruction single-data fashion, according to Flynn's Taxonomy[36]. They typically consisted of a memory storage system and a processing unit. This is known as the Von Neumann architecture[118]. Flynn's Taxonomy came about when vector processing was invented. Figure 2.2 illustrates the taxonomy and shows typical implementations of each category.

Vector processors were conceived in the early 1960s: these allowed mathematical operations to be performed on a vector of data instead of a single item of data. Early versions of this technology had potential, but the first widely accepted high performance computing machine released was the Cray-1[23] in 1977. Vector processing was routinely implemented as a pipelining processing unit, which effectively processed a vector of data instead of replicating multiple floating point units. Many vector processing capable supercomputers were designed and implemented up until

the late 1980s.

Until this time, computers were designed as shared memory processing machines which were typically multiple processing units connected to several banks of memory with an internal interconnect. This allowed for global access to memory from each processing unit. These individual processors consisted of several sub-processing units, such as a floating point units or vector units, and typically involved more complex architecture such as pipelining.

The first generation of distributed memory computers designed were the Transputer machines in the 1980s. These featured dedicated memory per microprocessor with serial communication links connecting multiple microprocessors together. An early example of a Transputer was the Meiko Computing Surface[80]. Individual Transputers were single-instruction single-data (SISD), but were designed as a building block of larger scale MIMD computers and included dedicated communication hardware and instructions, as well as hardware thread scheduling. Transputers were widely seen as the next generation of supercomputer, but in the early 1990s massively parallel processing supercomputers were introduced.

Transputers were quickly eclipsed by higher performing general purpose microprocessors. Increased performance and support for more traditional programming models outweighed the advantages of the transputer. Later parallel systems made greater use of mainstream microprocessors and networking hardware, though some systems continued to use transputers as part of the networking layer for some time.

2.4 Recent History of Supercomputing

With the emergence of massively parallel computing machines came the introduction of networks within a single computer. Early examples of such computers were the Intel iPSC/1[63] and the nCUBE 10[49]. Both of these machines had networks with the hypercube topology as illustrated in Figure 2.3.

Hyper-cube topologies are best suited for small to medium sized networks, which is likely the reason they were chosen as early candidates[75]. The topology is limited to a power of two count of nodes, which becomes a disadvantage when larger machines are involved. Additionally, many algorithms which involve pairwise exchange of information map easily to a hyper-cube given the n-dimensional peer linking. An example of this is discussed in Chapter 5. The routing within a hyper-cube is an attractive feature, since finding a path between nodes is a series of xor-operations, which

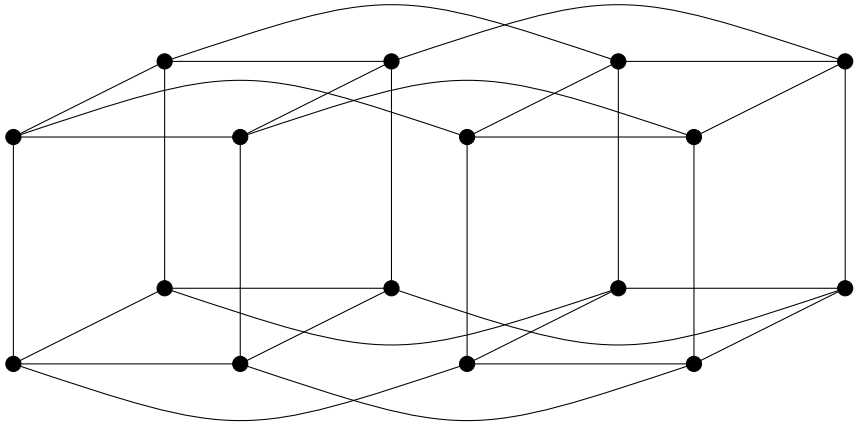


Figure 2.3: Hypercube topology of sixteen nodes. The degree of each node is $\log_2 N$ in a hypercube topology which makes it infeasible to use for large systems.

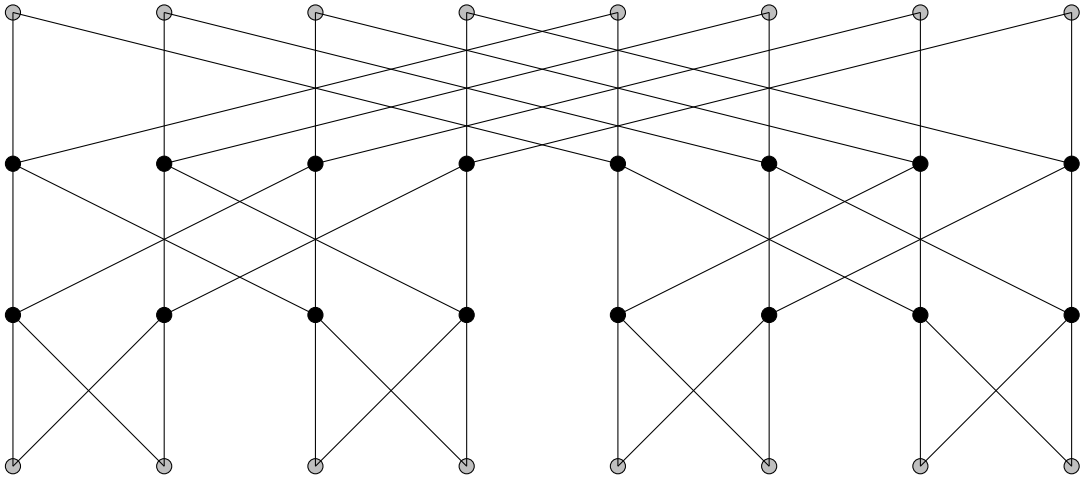


Figure 2.4: An example of a butterfly topology with eight nodes. The grey vertices represent the same switch position in a wrapped butterfly topology.

lends itself to hardware implementation.

Another example of an early network topology used was the butterfly topology, illustrated in Figure 2.4. This topology was never a popular choice and few computers were designed to take advantage of it. The advantages of the butterfly topology are a lower diameter than most topologies, matching hyper-cubes, and a high bisection bandwidth. Despite these benefits the complexity of the butterfly topology is the main restriction in its use.

The Cray T3D marked the first massively parallel distributed memory architecture from Cray using an interconnect and up to 2048 processing elements[1]. The interconnect was a three dimensional torus topology as shown in Figure 2.5. The Cray T3D

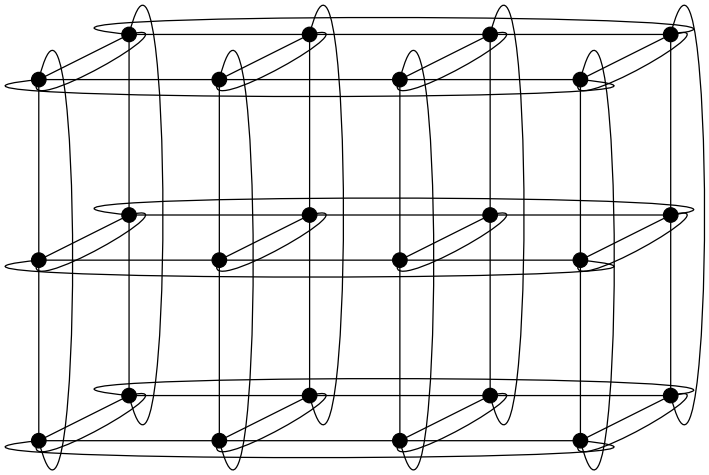


Figure 2.5: An illustration of a three dimensional periodic torus topology with twenty four nodes. The n-dimensional analogue extends each edge node with an additional edge to the corresponding periodic peer node. The periodic edges show the connectivity of edge nodes. The degree of all nodes is equal across all nodes to twice the dimension.

bundled two processing elements together to form a node. A 6-way switch per node facilitated the network topology.

Each Processing Element (PE) consisted of a single CPU and attached memory. The network interconnect presented as non cache-coherent shared memory in that a PE was able to map the memory of any other PE into its own address space and read/write data using normal read/write instructions. This was an early example of Remote Direct Memory Access (RDMA). Unfortunately, the lack of cache coherency and the significant additional memory latency (especially for read operations which required a network round-trip for each instruction) meant that normal shared memory programming techniques could not be used. Instead, these systems were programmed using message passing or early Partitioned Global Address Space (PGAS) programming models that were largely developed in response to the T3D architecture. With PGAS the explicit synchronisation steps in these models could include cache flushes to work around the lack of cache coherency.

The Cray T3D follow-on system, the Cray T3E, also provided RDMA communication, though via memory mapped hardware rather than direct CPU read/write instructions as this allowed greater latency hiding.

The torus topology is beneficial for the Cray T3D, because it allows a larger node count than either butterfly or hypercube topologies. Another advantage is the ease of expanding, since one does not need to reconfigure the entire topology. However, compared to butterfly or hypercube networks, the number of hops is increased on average. In addition, the cost of the wiring of a torus increases compared to earlier topologies.

At the beginning of the 1990s software was typically written to be platform specific. Porting software to multiple platforms, which is intrinsically a beneficial property of software, was difficult and expensive due to varying exposed capabilities of those platforms. At this point it became much less common for application codes to to take account of specific details of the network hardware or topology. These became the concern of the MPI library developers, while applications were written assuming any process could communicate with any other in a symmetric fashion.

2.4.1 Message Passing Interface

MPI is the *de facto* specification of the message passing parallel programming model which is present in modern high performance computing. The message-passing programming model is defined by multiple processes cooperating in parallel without shared

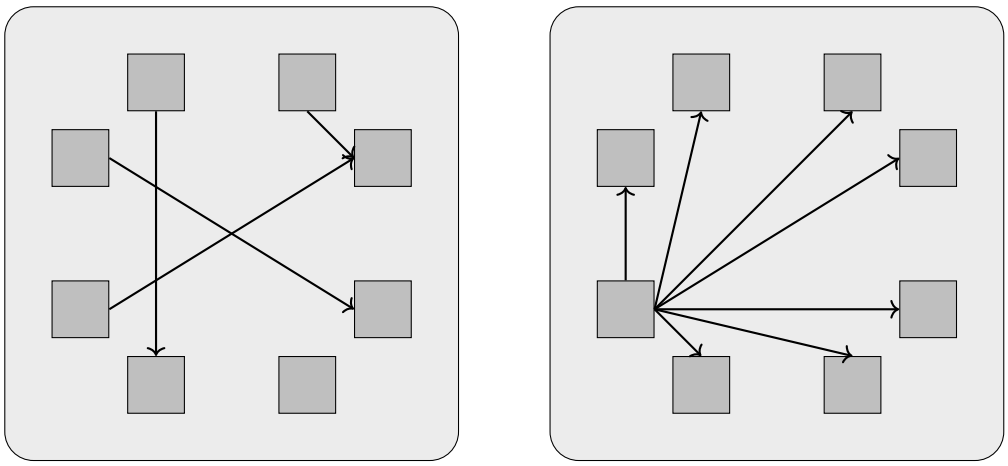
access to each other's memory spaces, but being able to communicate and synchronise by sending messages. The MPI 1.0 specification[85] was introduced in 1993 by a panel of computing centers and industry partners. MPI founded a unified approach to message passing. Previously several standards were present, implementing message passing on separate platforms.

Examples of these were Intel NX, PARMACS[14], Zipcode[104], PVM[42] and IBM EUI/CCL. These included most functionality present in the early specification of MPI, such as two-sided message passing and collective operations. These libraries did not provide all functionality, which therefore triggered the creation of MPI. Further development was required to produce a cross platform application, due to the non-overlapping functionality exposed by the libraries. A central theme in MPI is communicators, introduce in Bruce et al.[12], which act as communications domains which are entirely separate from other communicators.

Several communication interfaces are available to be used through MPI. The point-to-point functionality introduced in MPI 1.0 provides simple peer to peer communication through several interfaces, providing blocking, non-blocking and persistent operations. By allowing the user to select between standard, buffered, ready-send and synchronous modes it was possible to optimize for all platforms of the time. In addition, non-blocking forms of these functions were provided to allow for computation-communication overlap. Figure 2.6a shows an illustration of a group of processes communicating via point-to-point operations within a communicator.

The collective operations in the MPI specification are more independent of the user than the point-to-point operations. Collectives are operations which are executed by a group of MPI processes which coordinate the entirety of the operation. MPI collectives include data movement operations, reductions and synchronization. As collective operations are high level, they provide the library developer with an opportunity to optimise for specific network hardware or topology without requiring any changes to the higher level application code. Figure 2.6b shows a broadcast within a communicator in which all processes participate.

MPI exposes a third set of communication routines under the category of one-sided operations named MPI RMA. This functionality was added to MPI-2.0 due to the popularity of RDMA networks and the ubiquity of MPI usage. The Remote Memory Access (RMA) functions provide an interface similar to what would be found in a PGAS model implementation. Functions for data movement are exposed as PUT and GET operations. In addition a reduction operation and synchronization is present, but



(a) MPI Point to Point (b) MPI Broadcast Collective

Figure 2.6: An illustration of the difference between point-to-point communication and collective communication. Collective communication allows regular patterns to be described as part of the communication task; whereas point-to-point communication narrowly defines where and how data is moved.

no other collective operations. Access is handled by using epochs, which are periods of time of access to windows of memory which are exposed by peers.

Aside from explicit communication mechanisms, MPI implements higher-level abstractions which allow the programmer to program an abstract model of processes instead of a specific machine. MPI Datatypes are used to represent structures or data patterns in arrays. In addition Datatypes allow the MPI library flexibility in transmission of data across processes instead of having an explicit set of instructions from the programmer. The organization inside a communicator is a simple list of processes, but the programmer can specify a topology which can be used to decompose the problem and map it to the topology.

2.4.2 PGAS Interfaces

MPI is a well established standard which has extensive support and is used on all modern supercomputers. The PGAS programming model has not had as much time to be accepted, due to the relatively new hardware capability. PGAS is defined by exposing a global address space accessible by all participating processes with the concept of locality of memory to each process, and thereby also implies cost, in time, of accessing remote memory. Many different forms of languages and libraries exist which implement parts or all of the PGAS programming model.

Using language implementations of a programming model can have advantages over using a library implementation, because more information is able to be passed along to the compiler. The disadvantage of using a language and compiler infrastructure is the need to implement and support that infrastructure which is a significant cost. Usually these PGAS languages take the form of language extensions, such as Unified Parallel C [17] or Co-Array Fortran [89]. Both of these languages implement a pure PGAS model on top of RDMA hardware which can be found in supercomputers. Asynchronous Partitioned Global Address Space (APGAS) is an extension of PGAS, in which process spawning functionality is part of the model. Examples of APGAS are Chapel [19] and X10 [31].

While languages are good for capturing additional information which can be exploited by compilers, the problem is that usually compilers are not capable of this. In addition, libraries allow more fine grained control, which can be advantageous. Since PGAS is not a well defined concept, libraries contain their own approach. OpenSHMEM and Cray DMAPP, discussed in Section 3.3, are PGAS implementing libraries. In addition the MPI RMA operations in MPI-2.0 are also a PGAS approach. Finally a dedicated PGAS programming model specification exists which is called the Global Address Space Programming Interface (GASPI)[41].

GASPI contains the standard functionality expected from a PGAS implementation. In addition, the specification also provides groupings similar to MPI communicators which are used for collective operations. GASPI also provides, like Cray DMAPP, a put and notify functionality which allows combining two ordered put operations into a single compound operation. By using GASPI, the hope is to allow for more scalable applications on large scale machines.

2.5 Today

MPI has evolved alongside the hardware since the first massively parallel supercomputers were introduced. Several major shifts in the hardware have taken place since then. The first is the decline of Dennard Scaling.

Dennard scaling, which was responsible for much of the growth in computational capability in the past as discussed in Section 2.3, started to fail in the early years of the twenty first century. With the loss of Dennard scaling the inherent increase in clock frequency of processing units disappeared and performance improvements had to be gained by other means. This was particularly evident when Intel decided to focus on

multi-core processors in 2004, with the Pentium 4 marking the last fully single-core processors developed by the company.

By utilizing greater miniaturization of integrated circuits provided by Moore's law, CPUs were able to be created which contained multiple processing cores, linked with an internal interconnect. In addition, much of the memory management and expansion bus were moved onto the CPU.

While Moore's law has been said to be declining for several years, this has not yet happened in a material way. Hardware architectures are facing new challenges which will have to be overcome with a combined strategy of both software and hardware. The software must be adjusted to take greater advantage of the hardware and utilize it more effectively.

The current goal of the high performance computing community is to reach the Exascale computing era. Current supercomputer architectures tend to be built using large nodes with a modern interconnect to support inter-node communication. The large nodes consist of multiple CPUs which are multi-core up to the tens of cores. In addition to general purpose processing, heterogeneous processing utilizing GPUs or other accelerators has become the dominant strategy.

Modern networks support many programming models, but many operate on a RDMA basis in the hardware. This facilitates discrete distributed memory machines which can be programmed with a variety of network abstractions, such as MPI or PGAS frameworks. The number of nodes in modern massively parallel computers has grown to be approximately one order of magnitude greater than the first massively parallel computers, from the hundreds to thousands of nodes. The large increase in computing performance is achieved mostly by utilizing the accelerators and/or multi-core processors.

High performance computers were always shared resources, even during the early phase of computing (discussed in Section 2.2). Initially they were time shared machines which would be used sequentially by different users or programs. Eventually, this included splitting the machine into chunks, which resulted in a complex scheduling task for the job scheduling systems. Another effect of this splitting was that topology awareness became less important compared to the early massively parallel computing machines. An allocation given to a job run of an application is typically not regular and jobs rarely require an entire supercomputer to run.

In effect this made the topologies irrelevant by giving each application instance a unique and complex communication graph within which it must optimize its own

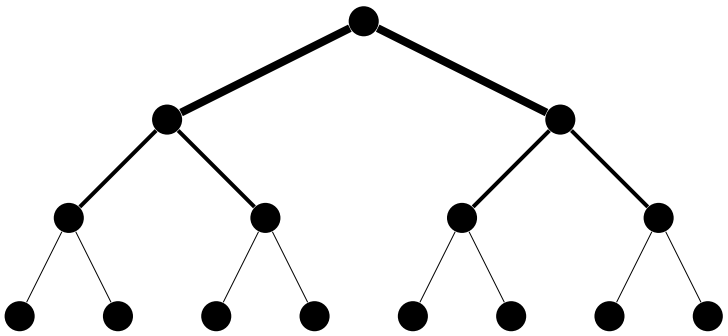


Figure 2.7: A fat-tree topology of eight leaf nodes. The thickness of the connections between nodes represents the relative bandwidth of that connection.

computation. With this shift in the networking infrastructure, hardware vendors moved towards more intelligent interconnects which handle congestion and perform adaptive routing of traffic within the topology accordingly. This isolates the complexity of networks and allows applications to view a network as a flat all-to-all topology ideally. The toroidal network topology introduced in the early era of massively parallel supercomputers has faded and been replaced by either the fat-tree topology[76] or the dragonfly topology[69] discussed in Chapter 3. Figure 2.7 illustrates the fat-tree topology.

Due to the abstraction of the network, topologies are no longer as prominent as they once were. Modern interconnect technologies utilizing these topologies delegate much of the network complexity onto the Network Interface Controller (NIC), or into the network, while only providing high-level access to a user-level library or application. Another feature of modern networks is the use of multi-rail NICs, which provide multiple routes into the network. The multi-rail aspect can give additional benefits for algorithms which execute network operations.

Similar to the hardware complexity, software complexity has also increased since the early days of computing. Applications were written *close to the metal* early on, but, when portability was prioritized, abstraction layers had to be introduced. MPI is one of these layers, however both above and below MPI many additional layers exist.

Above MPI, libraries exist such as HDF5[37] which is a commonly used hierarchical data format that operates across nodes, using MPI File IO as a provider of network communications. The Fastest Fourier Transform in the West (FFTW)[39] library is another example of a abstracted library. It provides the Fast Fourier Transform algorithm as a library which, for distributed memory architectures, uses MPI extensively. With many applications, developers write internal abstraction layers to allow for encapsu-

lation within the software. Finally Charm++[67] is a programming language which is used to write several applications, but can be implemented on top of MPI.

Below MPI, typical implementations directly access the hardware Application Programming Interface (API), but in recent years this has changed as well. Much of the hardware has internal software layers which execute without the direct awareness of MPI. In addition, UCX[102] and Libfabric[48] have been introduced to create a uniform interface to target from the MPI layer.

In summary, we can see that MPI plays an important role in the software stack present on modern supercomputers. MPI fulfills a middle-ware role by providing a long accepted interface and certain convenience functionality for higher level operations such as collective operations, file input/output and other higher-level functionality. Other libraries which require communication are built on top of the MPI to enable portability.

Chapter 3

Experimental Setup

3.1 Introduction

To discuss collective operation algorithms we need to have a good understanding of the underlying hardware. By understanding the hardware on which we operate we can explore the space of possible solutions. In addition, an understanding of the abstraction layer, MPI in our case, is also required since the goal of this work is to fulfill the requirements for the algorithms to be used by an MPI implementation.

This chapter will discusses the hardware and software on which experiments have been performed throughout this work, except where mentioned. The aim of this chapter is to inform the reader about the capability and accessibility of compute and communications hardware.

In this work we focus on the Cray XC Series, but the methods and algorithms described are equally applicable to modern Infiniband networks. This is shown by recent work performed by End et al.[34] which makes use of network pipelining effects as well. In addition, future networks such as HPE Cray Slingshot[26] show little improvement in latency, but significant improvements in bisection bandwidth and injection bandwidth. Combined with the capability of having multiple NICs per node future networks are not expected to perform less overall message pipelining.

Section 3.2 discusses the Cray XC30 computer hardware and specifically Section 3.2.1 goes into detail about the Aries NIC. Section 3.3 introduces the available APIs of the given hardware for communication; the discussion focuses on DMAPP due to the work performed with this particular library.

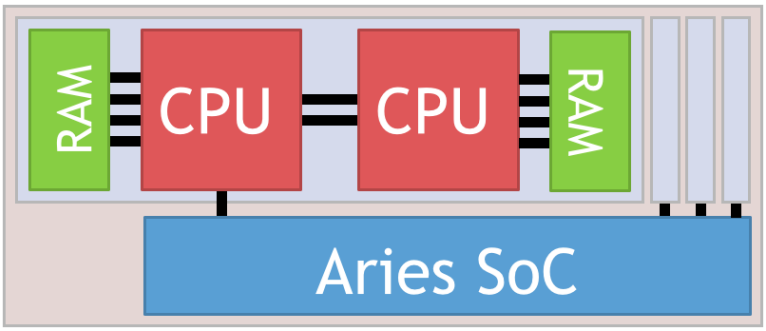


Figure 3.1: Illustration of a blade within a Cray XC30 system.

3.2 Hardware

Modern supercomputers such as the Cray XC30 are very large and extremely parallel machines. While supercomputers were always parallel to a degree, since Dennard scaling[28] has broken down, industry has been focusing on multi-core processors. This has lead to a large increase in parallelism in all computers from desktop machines to supercomputers. As an example, the maximum size of the Cray XC30 is 92544 nodes, which would allow in the order of one hundred peta-FLOPS. Currently the largest Cray XC30 supercomputer is Piz Daint with 5272 computer nodes using Tesla K20X accelerators. Piz Daint delivers a theoretical 7.79 peta-FLOPS[106].

The Cray XC30 using Intel CPUs is organized in a hierarchical fashion. The smallest unit of computing hardware is the node, which consists of two sockets connected by the Intel Quick Path Interconnect. Both sockets allow for a multi-core CPU. Each node also has either 64GB or 128GB of shared local random access memory accessible by the processors. Each node is organized in the next smallest compute unit, the blade. A blade consists of four nodes which are connected to an Aries System On a Chip (SOC). The Aries SOC consists of four network interface controllers for the nodes, a tiled router and a multiplexer node. Figure 3.1 illustrates the structure of a compute blade with relative bandwidths illustrated between separate components.

Many blades form the Dragonfly topology, with three separate layers of organization. A set of sixteen blades are organized into a chassis, with all blades being connected to each other over an electrical back-plane in an all-to-all pattern. The chassis connectivity in the Cray XC30 is considered the rank-1 network. The rank-2 network is the set of connections which is referred to as cascade. The cascade pattern connects six chassis to form a group. The cascade pattern connects three links from every blade in a chassis to a corresponding blade in one of the other five chassis. This also

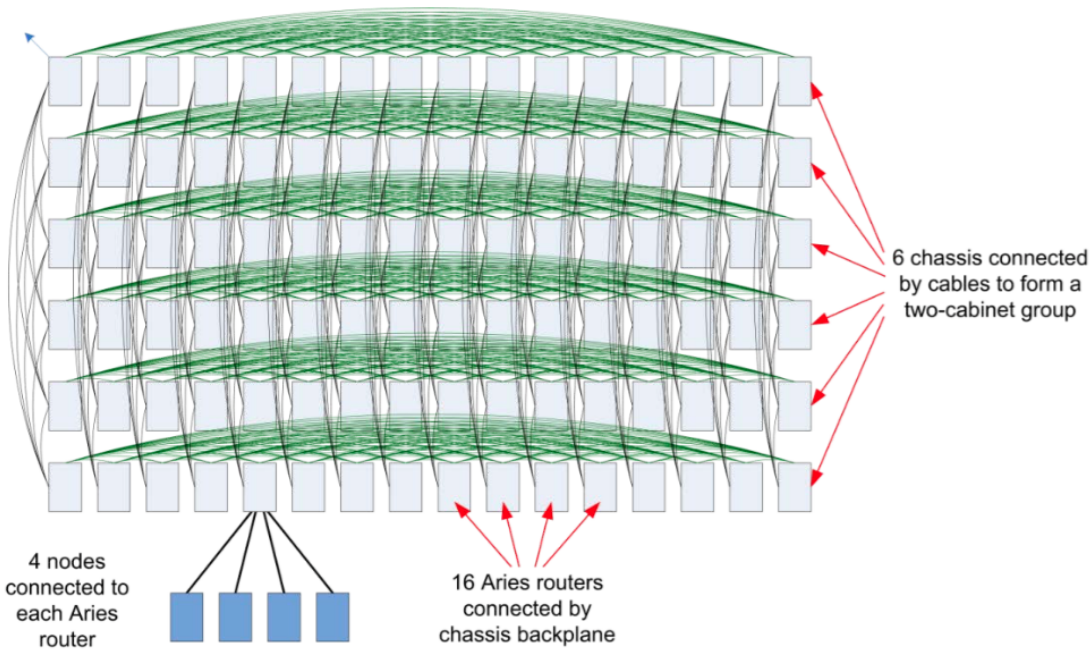


Figure 3.2: Cray XC30 rank 1 and rank 2 connectivity illustration.[3]

forms an all-to-all topology between the chassis in a group as illustrated in Figure 3.2. The final rank-3 level of the Dragonfly topology is an all-to-all pattern between groups using global optical connections, as illustrated in Figure 3.3. Utilizing this topology, the minimal path for a packet crossing the entire machine is at most five hops. However, in a live machine this is likely to be shifted upwards due to adaptive routing and congestion.

The Cray XC30 operated by the Edinburgh Parallel Computing Centre (EPCC), named ARCHER, was used exclusively for this work. ARCHER at node level uses two 2.7Ghz, 12-core Intel Ivy Bridge E5-2697 v2 central processing units with either 64GB or 128GB of random access memory: other memory configurations are possible for XC30 machines. In total ARCHER consists of 4920 compute nodes connected in 13 groups in a Dragonfly topology. No accelerators, such as graphics processing units or Intel Xeon Phis, are present on the ARCHER platform.

3.2.1 Aries Network Interface Controller

The Aries SOC consists of several parts. However, the NIC is of primary interest when concerned with functionality usable by an application. The routing is entirely transparent to the programmer, but the user can choose between four routing modes. The Aries NIC has several functional units which allow the Cray XC30 to perform low

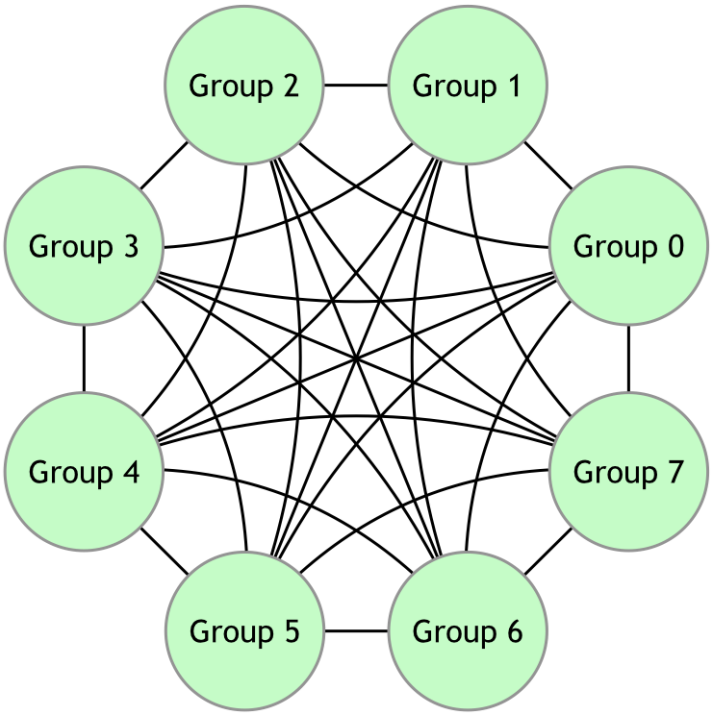


Figure 3.3: Cray XC30 groups all-to-all connectivity illustration.[3]

latency and high bandwidth operations on large systems.

The Fast Memory Access (FMA) mechanism, which allows for small sized, low latency access of remote memory, is exposed through an API. In addition the Block Transfer Engine (BTE) is provided for large memory copies. The BTE can execute entirely asynchronously to the host processor, unlike the FMA unit. Both of these units provide the RDMA functionality which enables access to remote memory without interfering with the remote host processor. Both the FMA and BTE unit utilize the completion queue system present in the NIC to inform either the remote or local processor of operation completion.

The FMA mechanism allows for direct memory access through PUT and GET operations. Aside from memory access, the FMA mechanism also provides Atomic Memory Operations (AMOs). There are a total of 96 AMOs provided by the Aries NIC: these include bit-wise operations, integer operations and floating point operations, with both fetching and non-fetching semantics. In addition the Aries NIC has a small 64-entry AMO cache which caches a local copy of a memory location. The cache is primarily used to reduce the host memory usage when the automatic write back functionality is deactivated.

Due to the use of virtual memory address space on modern supercomputers, the

Aries NIC is also required to deal with virtual to physical memory address translation. This is done similarly to a processor using an Input Output Memory Management Unit (IOMMU). The IOMMU uses a table of registered pages saved in memory and caches these in a page table cache. This unit functions equivalently to a CPU Translation Look-aside Buffer (TLB). Multiple page sizes are supported from 4KB up to 64GB, but 4KB pages are supported via two-level translation, while larger page sizes are single-level translation. The usage of huge pages is encouraged, due to the large cost of a cache miss. The cache size is 128 entries, each containing eight page blocks, using a four-way set associative caching strategy.

The BTE unit is explicitly put in place to allow for offloading the transferring of data from one node to another. The BTE is used for large messages exclusively by default above four kilobytes. The FMA transport mechanism is specifically designed for short messages and therefore does not provide any exposed pipelining capability. Through Cray DMAPP, discussed in Section 3.3.2, it is possible to send small messages using an implicit synchronization mode which allows overlapping of small messages as well. This mechanism is further discussed in Chapter 4 and used in Chapter 5.

3.3 Software

The Cray XC30 provides several user-level libraries or languages to interface directly with the low-level network Aries-based network. As seen in Figure 3.4, the user-level libraries mostly bypass the kernel-level complexity to ensure high performance, interfacing instead directly with the hardware abstraction layer. Several programming languages are presented in Figure 3.4, such as UPC (which will not be discussed further here).

3.3.1 Cray Shared Memory

CraySHMEM is a one-sided memory access library, developed originally internal to Cray in 1993. Historically it was first used to program the Cray T3D, and later adapted for use as a programming model for distributed memory clusters as a low-level interface for languages. OpenSHMEM[90], which is an open standard, has now superseded CraySHMEM. OpenSHMEM provides a more diverse set of functionality, such as atomic operations and collective operations in addition to the low level memory operations.

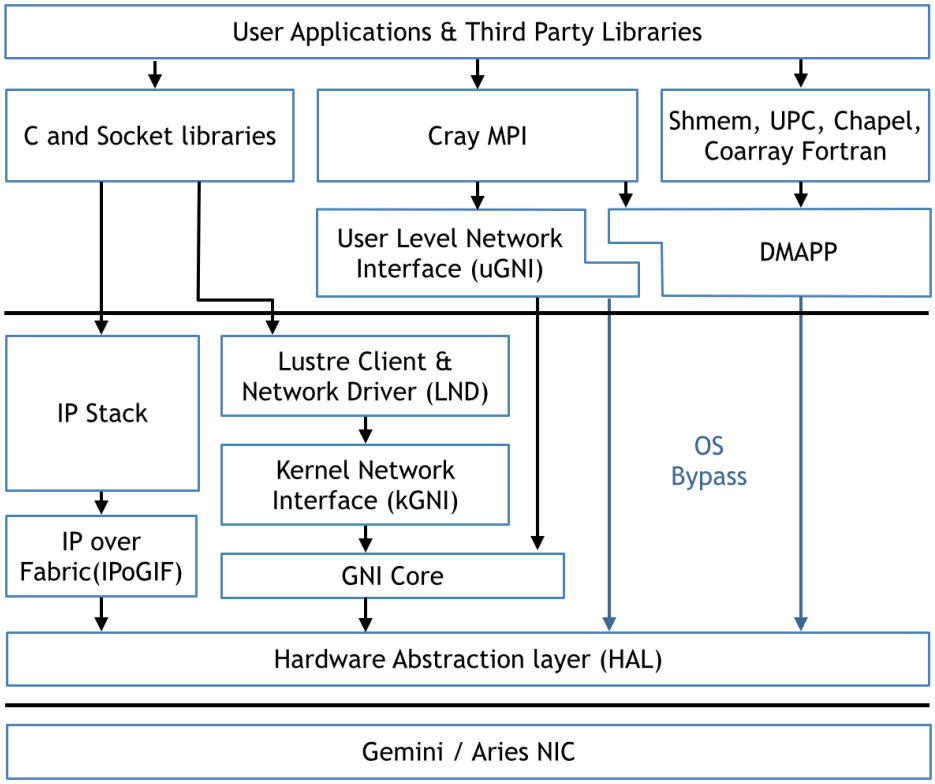


Figure 3.4: Software stack present for the Cray XC systems[3].

3.3.2 Cray Distributed Memory Applications

Two main parallel programming models are used to program large distributed systems such as the Cray XC30. The PGAS model is newer, and used less frequently, compared to message passing. The PGAS model, when implemented through languages, depends strongly on having a good compiler which can interpret the communication performed through memory transactions and translate them to instructions which can take advantage of the underlying hardware. Cray DMAPP is the low-level networking library which is used on Cray hardware for this task. DMAPP is specifically designed to deal with the PGAS model. It provides functionality which supersedes CraySHMEM (presented in Section 3.3.1) and therefore one could view it as a successor.

Cray DMAPP is designed in such a way that multiple processes run on separate nodes executing the same program concurrently in a Single-Program Multiple-Data (SPMD) fashion. All processes have their own address space, as full operating system processes, but it is possible to publish access to memory segments from the private address space. This is achieved through memory registration, a major feature not present in SHMEM, where all memory is public at all times. In addition, DMAPP allows access to the hardware features provided by the Aries NIC to communicate on a low-level

over the network. Only simple abstraction is present, which hides some of the difficulties of using hardware directly, such as the completion queues discussed in Section 3.2.1.

The functionality provided by DMAPP is similar to SHMEM as a one-sided memory operations library. This include PUT and GET RMA operations, AMO operations (with a limited set of AMOs exposed to the programmer), and collective operations which make use of the Aries NIC collective engine. In addition, DMAPP provides a symmetric heap at initialization which allows SHMEM-like operations to be performed. The symmetric heap must be used to establish any communication, since asymmetric memory cannot be accessed without the segment information received from DMAPP when registering. A unique addition to DMAPP is a queue subsystem which allocates a queue at initialization and allows a user to attach a callback function to process an incoming message from a remote process. These queues are dynamically allocated with a fixed size in symmetric memory, which allows the parameters to be used to control the size of the queue.

Cray DMAPP is a low-level API and therefore does not provide the level of functionality that MPI does. The most simple synchronisation possible on a Cray machine is to use a global barrier provided by Process Management Interface (PMI), a low-level process management library. The global barrier is usually useful for the initiation procedure, however it would rarely be used within the execution flow of an application.

Since Cray DMAPP is a single-sided interface for communication, there are two different aspects to synchronisation. On the sender-side the API provides blocking, non-blocking explicit, and non-blocking implicit versions for most communication functionality. The blocking calls result in a safe transaction without any concern about memory reuse. Non-blocking functionality requires waiting for completion. This is very similar to the MPI notion of blocking and non-blocking. There is no equivalent persistent operation mode. The explicit non-blocking mechanism allows the user to hold individual handles to synchronisation identifiers, which allows the user to complete separate communications individually. The implicit non-blocking mechanism delegates this to Cray DMAPP, which then waits for all the implicit handles with a single waiting operation.

For the receiver-side synchronization, three potential methods are present depending on the requirements. The first is using a *put and flag* operation which completes the put operation and then sets a flag on the remote side, which serves as a signal to the receiver-side that a *message* has been placed into its buffer. A more explicit syn-

chronisation mechanism can be created using a variety of atomic memory operations: the most common for such a case is the *compare and swap* operation. Finally, Cray DMAPP supports creating process sets, akin to communicators within MPI, which can be operated on by collective operations. The collective operation primitives given in the API are barrier and AllReduce.

The above discussed mechanisms are the lowest-level synchronization methods in place within Cray DMAPP. Two additional higher-level mechanisms are implemented: the first is the explicit lock system. The lock system allows a calling thread to lock one of the many available receiver-side locks in order to have exclusive access to a memory segment. The second is a high-level single-consumer multi-producer (SCMP) queue implementation. The queue mechanism requires a user provided callback function which handles the incoming messages. It also allows either a separate asynchronous progress thread or for the user to call the progress function.

3.3.3 Cray user-level Generic Network Interface

The user-level Generic Network Interface (uGNI) shown in Figure 3.4 is the lowest level API which Cray presents to a third party programmer. It is designed to complement the Cray DMAPP implementation by providing functionality which is more relevant to message passing than utilizing the RDMA functionality of the Cray Aries network. It is primarily used for implementing MPI, which is discussed in Section 3.3.4. In addition to facilitating access to the Aries NIC, it also exposes datagrams, which are the lowest level communications path present through uGNI, but not through DMAPP.

3.3.4 Cray MPI

The Cray MPI library is provided with all Cray hardware through the Cray Message Passing Toolkit. The Cray MPI implementation is known to be based on the MPICH2[47] library implementation of MPI[91]. Many vendors implement derivative libraries from either MPICH2 or OpenMPI[45] for their hardware and dedicated support versions of MPI due to the cost and time requirements of implementing a full native version, which is prohibitive. MPICH2 is a descendant of the reference implementation of the previously discussed MPI Standard in Section 2.4.1. The only publicly documented addition to the reference MPICH2 implementation is the Nemesis network module, however it is likely that changes have been made to several parts

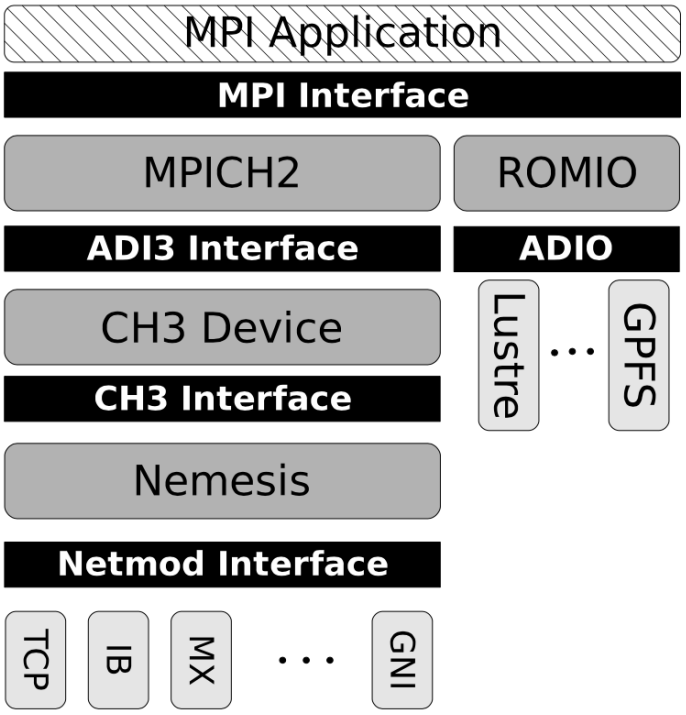


Figure 3.5: The MPICH2 layer diagram showing the internal layering of the channel interfaces[92].

of the implementation.

Figure 3.5 illustrates the structure of the MPICH2 library. The library provides the entire MPI specification functionality, but implements it through various internal abstraction layers. Within the core layer of MPICH2 implementations for collectives, matching algorithms and other top-level functionality is contained. The network interaction is delegated further downward to devices which represent the underlying hardware.

The Nemesis[13] communication subsystem implements the CH3 interface to communicate upwards through the CH3 device. It implements lock-free queues, to allow for low latency small message operations with high bandwidth for large messages. The authors focus on reducing the instructions required to perform intra-node and inter-node communications. The additional abstraction below the Nemesis module is the Netmod interface with which the underlying hardware APIs are accessed. The Netmod for the Cray hardware utilizes the uGNI interface, discussed in Section 3.3.3.

Within the Cray Netmod, most facilities provided by uGNI are used. Connection setup uses connection-less datagrams between endpoints to facilitate the construc-

tion of endpoint connections which are used by the SMSG and MSGQ channels. The SMSG channel is used to send small eager messages directly to a process' memory. The memory size of the SMSG mailbox depends on the size of the job. The MSGQ channel is an optimization for memory usage which utilizes a mailbox per node instead of per process. This significantly reduces the memory requirements of large jobs.

The *E0* eager protocol is used when the entire message fits into a single mailbox of either the SMSG or MSGQ. For larger messages up to the eager size limit, the *E1* protocol is used which consists of an *E0* channel interaction and a FMA or BTE operation from the receiver to fetch a message into an MPI internal buffer. For messages above the eager message limit, the *R0* and *R1* paths are used through the Nemesis long message transfer path. The *R0* protocol is equivalent to the *E1* protocol except that the message is fetched directly into a user buffer. Finally, the *R1* protocol is a *PUT* operation by the sending side directly to the final user space buffer.

The Netmod utilizes the uDREG library, which provides a memory registration cache, for the large message transfer path. If an application is also using DMAPP directly, or through CraySHMEM, UPC or Coarray Fortran, then the Netmod will use DMAPP to register memory regions. This allows DMAPP and uGNI to share memory registration resources.

Chapter 4

Performance Modelling

Performance modelling is a core aspect of algorithm design: it functions as a proxy of a real-world machine and simplifies the characteristic dependencies. Many performance models exist, but few attempt to represent all aspects of a supercomputer. We introduce the pipelining latency-bandwidth model to represent the small message pipelining capability of modern supercomputers, showing a higher accuracy from this model than from the latency-bandwidth model. In addition, we implement a discrete-event simulator to capture effects which are not analytically available.

4.1 Introduction

This chapter will discuss the various forms of modelling present in modern computer science. It presents the relevant historical and currently competing models, and will introduce our own model and simulation.

Theoretical models serve several purposes in computer science. Historically they formed the most fundamental understanding of the target with which software was written. Models can range from mathematical guides, to simplified representations of hardware, to a common design target between different layers.

The goal of models is to simplify and distill the essential characteristics of a computer, which are relevant to the specific topic that one is approaching. By using a model, a researcher can make abstract many details which are not critical to their specific component. This removes the *clutter* which would be present if all components had to persistently be taken into account: this would yield a far more complex forward path and therefore less productivity.

The first models used were the sequential machine models which gave algorithm designers an abstract view of the underlying hardware. The Random Access Machine (RAM)[21] model was an example of this, and continues to be used to this day. The RAM model is part of the bigger register machine family of models. The goal of these models is to form a common understanding and interface, such that both hardware and software can communicate through this abstract layer.

The Parallel Random Access Machine (PRAM) model[38] is an extension of the RAM model into the domain of parallel computation. The PRAM model is well suited for shared-memory machines, but networked communications were not well represented, due to assumption that all communication is transparent. It is important to note that models are typically closely related and any extensions introduce only minor changes in order to achieve representation of a specific feature.

4.1.1 Contributions

This chapter makes the following contributions:

- The pipelining latency-bandwidth model is introduced to represent small-message pipelining on modern supercomputers.
- The *Fennel* simulator is introduced, a discrete event simulator, focused on models and algorithms.

4.1.2 Overview

This chapter contains a discussion of relevant performance models in Section 4.2. We introduce our novel pipelining latency-bandwidth model in Section 4.3. Finally, we introduce the *Fennel* model simulator in Section 4.4, which is based on prior work simulates our pipelining latency-bandwidth model with novel contributions.

4.2 Prior Performance Models

This section discusses parallel computation models which are deemed to be of interest to the discussion in this thesis. Importantly, this is not an exhaustive discussion of computational models, either sequential or parallel. Maggs et al.[78] and Zhang et al.[120] survey existing models of parallel computation.

4.2.1 Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) model was used in the 1980s by Valiant et al[114]. The BSP model was the first parallel computation model which did not assume communication was free, unlike the PRAM model. In addition to the model, it introduced a category of applications, the bulk synchronous applications.

Flynn's Taxonomy, introduced in Section 2.3, in which the Multiple-Instruction Multiple-Data (MIMD) category encompasses most of modern computing was extended by the late 1980s to encompass the concept of programs instead of only instructions: the single-program multiple-data and multiple-program multiple-data categories were introduced[25]. The bulk synchronous parallel model similarly moves the modelling aspect from conceptually dealing with instructions to programs.

Bulk synchronous applications have an iterative behaviour which results in the series of global *supersteps* model of BSP. An application would perform a concurrent computation phase and then would perform a concurrent communication phase to communicate beyond the local memory space into the distributed memory of the other participating processes. Finally, an optional barrier synchronization is performed to finish the superstep.

Figure 4.1 illustrates such a *superstep* of an application with a varying computational phase, a communication phase and finally a barrier, given the communication is done over one-sided operations. The fundamental difference between BSP and previous models is that the communication phase is no longer transparent and has an

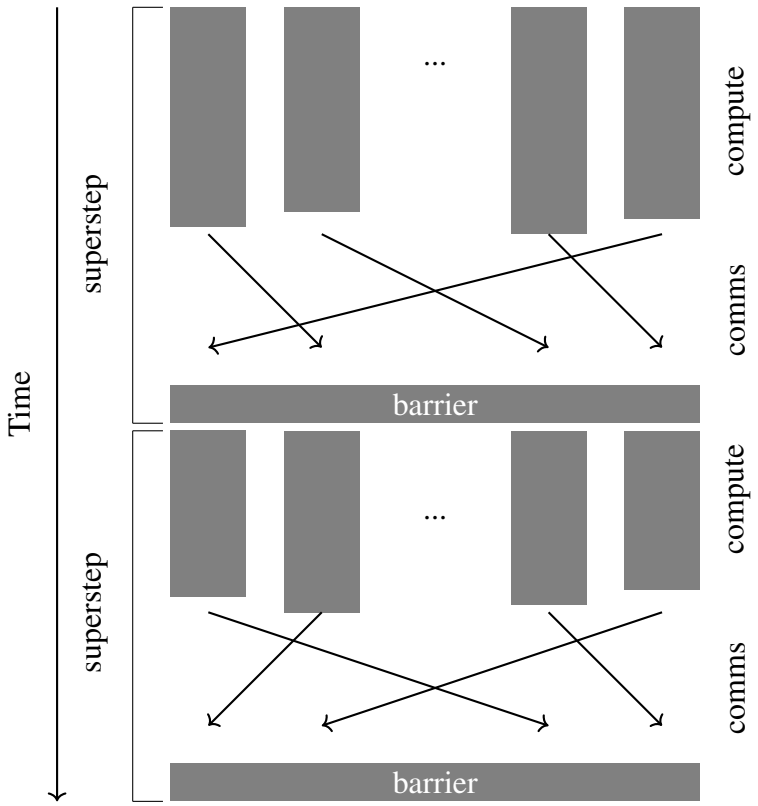


Figure 4.1: Two supersteps illustrated in the BSP model. The computational steps vary in time while the communication steps are always equal.

inherent cost correlating with the amount of communication performed.

Equation 4.1 is used to calculate the duration in the BSP model for an application with S supersteps, p processors, and a barrier cost of l. The variables used are ω_i representing the computational time of a single process within a single superstep, h_i representing the number of messages sent by a process, and $\bar{g}$ which represents the asymptotic throughput.

$$T_{\text{BSP}} = max_{i=1}^{p}(\omega_i) + max_{i=1}^{p}(\bar{g}h_i) + l \tag{4.1}$$

The communication cost within the BSP model is defined as a latency and a cost proportional to the total amount of data a process receives, the bandwidth term. They target their model at *optimal* simulations and choose to ignore the latency cost, asserting that the bandwidth term is at least as large as the latency. The underlying topology is ignored within the BSP model.

Since the initial publication of the BSP model, many extensions have been performed due to increased interest from large companies in addition to scientific high performance computing[9, 113]. Examples of implementations of the BSP model are

BSPlib[116] and MapReduce[27].

4.2.2 Latency-Bandwidth Model Family

The latency-bandwidth model family is characterized by the Equation 4.2, in which the parameter α represents the start up cost of a communication and β represents the time cost per byte sent. The message size to be transmitted is represented by n. The T_{LB} time duration starts when sending the message until the receive is complete. The model assumes that the sender is not able to continue until the receive completes. Importantly, this neglects any acknowledge messages which are sent to a sender from the receiver on modern interconnects.

$$T_{\text{LB}} = \alpha + n\beta \tag{4.2}$$

Typically these models also have associated rules such as invariance to topology, bidirectional linking, network conflict handling, and many other properties[6, 7, 20, 52, 53, 54, 82, 103, 109, 110]. These models have been used since at least 1977.

In addition to the basic latency-bandwidth model, if required, a compute component γ is added which represents the cost of computation time per byte as shown in Equation 4.3. In which case, the T_{LBC} represents the duration of a stage of an algorithm similar to the BSP model discussed in Section 4.2.1, since it includes communication and computation within the model.

$$T_{\text{LBC}} = \alpha + n(\beta + \gamma) \tag{4.3}$$

One commonality of this family is that communication and computation cannot overlap, because the CPU and NIC are treated as a single unit. When a communication operation is in progress, no computation can occur and vice versa. This lack of overlap led to the introduction of the LogP model, which we will discuss in Section 4.2.3.

In literature, the Postal model[5] is sometimes mistakenly associated with latency-bandwidth models. However the Postal model is more similar to the LogP models discussed in Section 4.2.3. The main motivation for the Postal model is to move away from telephone modelled communication to packet-switching with large messages being broken up into multiple smaller messages.

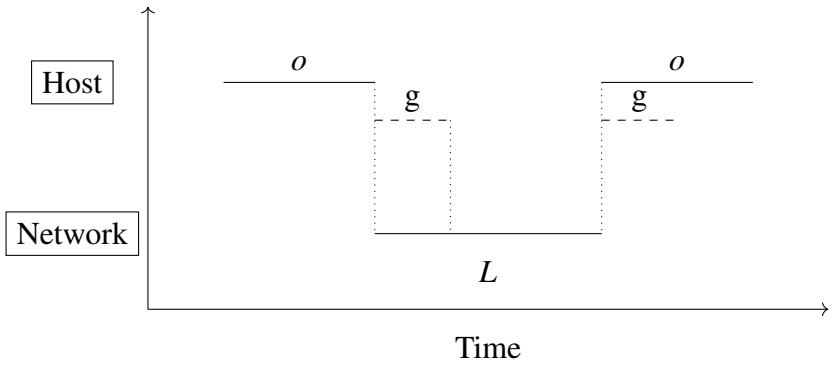


Figure 4.2: An illustration of the LogP model showing the transmission and reception of a message. The illustration is intended to show the path through the LogP model of a single message; therefore two hosts are drawn on the same *host* level, but are entirely separate physically.

4.2.3 LogP Model Family

The LogP model[24] was introduced as a convergence of computer architectures was observed in the early 1990s. Algorithms developed with prior models were routinely exploited analytically, in a fashion which was not grounded in reality. The LogP model sought to discourage such behaviour from researchers. The authors based the LogP model on the BSP model discussed in Section 4.2.1, but extend it to be asynchronous and to consider the network in more detail. They intend the model to be targeted at an intermediate level of complexity for distributed memory, compared to simplistic models such as PRAM, or overly complex machine specific models.

An illustration of the LogP model is shown in Figure 4.2. The model does not take into account the topology of the network. Instead, it simply uses abstract parameters to model the typical conditions on the network. The following parameters are used within the model:

- L — latency — the upper bound of the latency incurred for a small message (one word) between any two processors
- o — overhead — the delay incurred by a processor during which time it is occupied with sending or receiving a message
- g — gap — the time between consecutive message transmissions or receptions
- P — processors — the number of processors within the network available to the program

The model also includes the assumption of the network capacity being finite, such that at most $\lceil \frac{L}{g} \rceil$ messages can be in transit. The reciprocal of the gap parameter is the per-processor communication bandwidth. The computational aspect of the model is not explicitly stated, but typically it is modeled by a constant parameter as in Equation 4.3.

The LogP model has become the *de facto* parallel distributed memory performance model and has been extended in many forms, despite the original authors' intention to be more abstract.

The LogGP model[2] introduced in 1995 extends the LogP model with a G parameter. The G parameter is a measure of the communication gap per byte sent or received. This extension targets applications which require the transmission of large messages, not only word sized messages supported in the LogP model. The authors of the LogP model addressed long messages by proposing using two processors per node, but this was not widely adopted. Many other extensions have also been made to the LogP model which will not be further discussed here[15, 59, 62, 68, 77, 84].

4.2.4 LoP

The LoP model[58] was introduced by Hoefler et al., to address the introduction of Infiniband[61] compatible networks in the early 2000s. Infiniband networks introduce the concept of hardware parallelism or pipelining of messages, which cannot be represented well with the linear LogP model, given the single gap parameter.

The LoP model splits the overhead parameter from the LogP model into send-side and receive-side overheads. In addition, it introduces the two h parameters, also one for the send-side and one for the receive-side. The h parameters represent the amount of time required by the NIC to process the message. Finally, the g parameter of the LogP model is discarded. Figure 4.3 illustrates the LoP model.

A model for round trip time is the basis of the LoP model, which gives a highly accurate fit to observed data. The round trip time is calculated as a sum of pipelining, processing and saturation, as a function of the communication peers. This results in an expression for the round trip time with six parameters and ensures that latency of a message is dependent on the number of messages issued. The complexity of this model stems from the number of parameters, in addition to the non-linearity of the model.

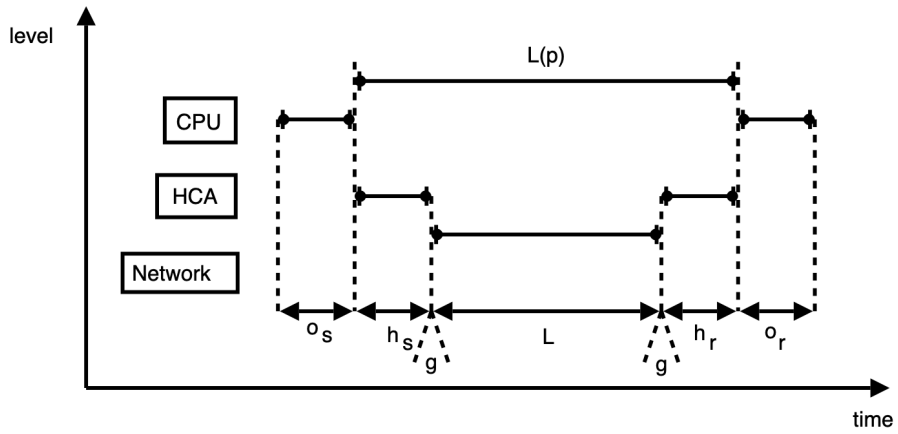


Figure 4.3: An illustration of the LoP model showing the transmission and receiving of a small message from [58]. The illustration is intended to show the path through the LoP model of a single message; therefore two hosts are drawn on the same levels, but are entirely separate physically.

4.2.5 LogfP

The LogfP model[57] is introduced to address the complexity of the LoP model. Instead of a complex model taking into account pipelining, processing and saturation, the model introduces the f parameter to the LogP model.

The f parameter represents the number of small messages for which the gap parameter does not apply. This suitably extends the LogP model to capture the effect of processing multiple small messages within the networking hardware. The LogfP also extends the overhead parameter o to be a function of the number of messages being sent.

Figure 4.4 illustrates the LogfP model sending four messages. As shown after f messages are sent the g parameter forces a delay sending further messages.

4.3 Pipelining Latency-Bandwidth Model

We introduce the pipelining latency-bandwidth model[98, 99] to address observed behaviour on the Cray XC30 discussed in Section 3.2.1. We wanted a model that could capture the behaviour of important collective communication operations such as AllReduce. In this case, the message sizes are usually small, so the time spent traversing the various network layers can be significantly longer than the time taken

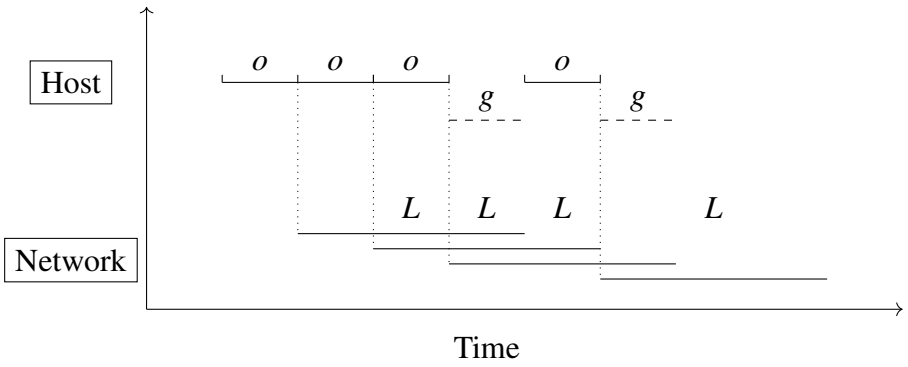


Figure 4.4: An illustration of the LogfP model using an f parameter of two. For the first two short messages the g parameter is not accounted for according to the model, but for successive short messages the g parameter applies. For simplicity the receiving-side is not illustrated.

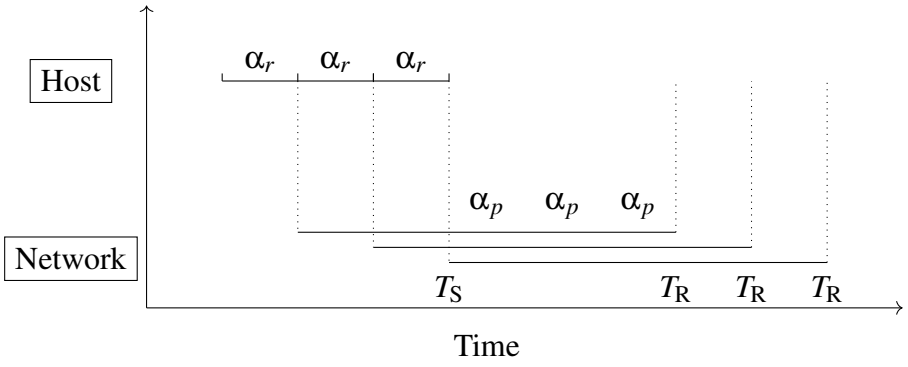


Figure 4.5: An illustration of the pipelining latency-bandwidth model setting the b parameter to three. A non-blocking multicast returns control to the host at time T_S, while a blocking multicast would return control after the last time T_R.

to process the message by either the CPU or the NIC at either end. In addition, we wanted to capture this behaviour with a small number of model parameters allowing us to fit our model to the observed behaviour of the Cray XC30.

The pipelining model splits the existing α parameter into the α_p and α_r parameters. The pipelining latency term α_p is able to overlap with other network operations, while the required latency term α_r is the equivalent of the o parameter of the LogP model: the time for which the host is blocked from doing any other work. Figure 4.5 illustrates the overlapping of messages being sent from a single host. One additional parameter b is introduced which specifies the number of messages which are sent.

Models presented in prior sections typically model either very abstractly, such as the BSP model, or they specify details about the underlying mechanisms with which

time is spent, such as the LogP model. Our model is introduced as an extension of the latency-bandwidth model, because it is sufficiently abstract to capture all point-to-point network operations without over specifying how time is spent.

The pipelining latency-bandwidth model uses two equations to represent communication time of a message. Equation 4.4 is used to calculate the time for which the sender is blocked when sending b messages in a non-blocking fashion. The duration for the message to traverse the network is given by Equation 4.5. If messages are sent with a blocking operation then the sender is blocked until T_{LBP_R} for the last message in the multicast. Since α_p includes any acknowledge messages this ensures that it is safe to reuse a buffer in memory by the sender. Using this methodology we overestimate the time at which the sender is able to resume.

$$T_{\text{LBP\_S}} = b\alpha_r \tag{4.4}$$

$$T_{\text{LBP\_R}} = \alpha_p + \alpha_r \tag{4.5}$$

The α parameters are measured directly from a host without abstract interpretation. With this approach the pipelining latency-bandwidth model represents *what the host sees*(i.e. the software interface) in local time instead of global time (which is not a physically possible thing to know[74]). The model is intended for small messages only, and is used as such in Chapter 5, but the model can be used with the bandwidth and computational terms included from Equation 4.3.

The representation of the software interface is a departure from what a model is traditionally used for. Traditional models attempt to represent the underlying hardware in as few parameters as possible to capture what is required, but the pipelining latency-bandwidth model purposefully forgoes this understanding to enable modelling of the software interface. Modelling the software interface allows capturing all hardware and software effects below a certain API instead of selectively treating each individual effect. In addition, it allows simpler measuring in local time since it is not reliant on global information.

The pipelining latency-bandwidth model follows closely the model presented in Chan et al[20] with similar assumptions about the properties of the abstract machine. We discard the receive conflict which disallows hosts to receive more than a single message at any given time. In addition, we ignore network conflicts, since the host has no control over these effects: they merely give a distribution to the given parameters.

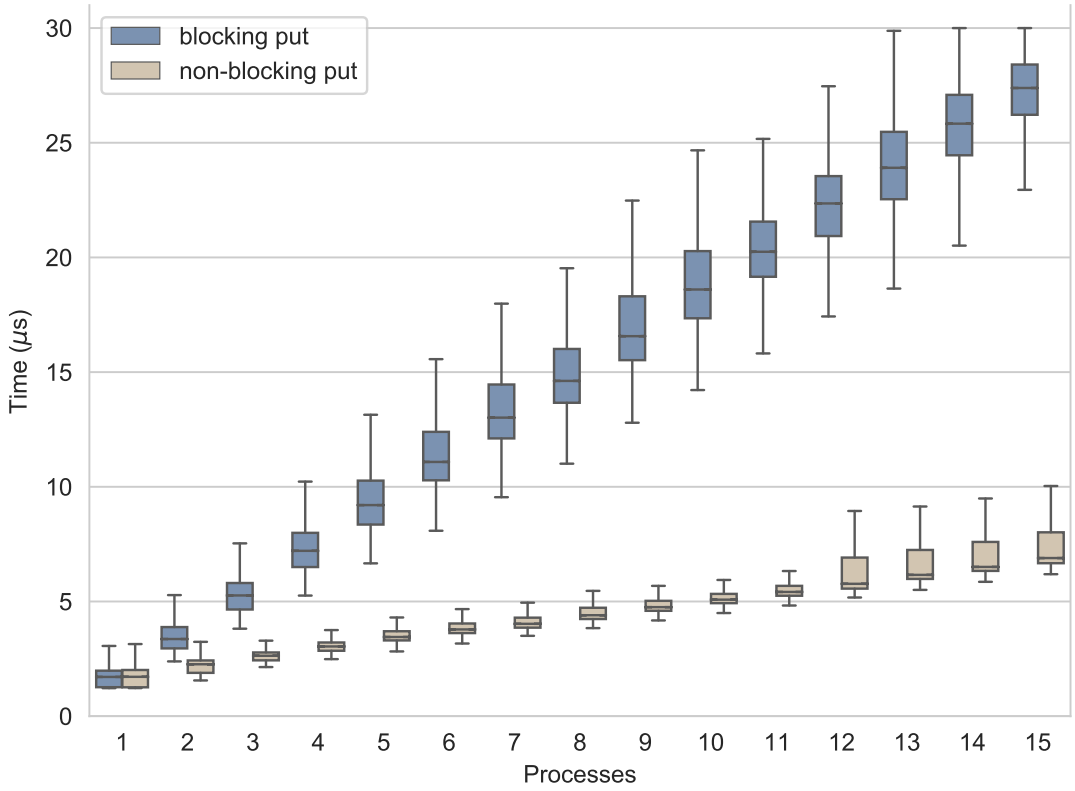


Figure 4.6: The plot shows the timing data of a sequential put operation to multiple target processes with either blocking or non-blocking semantics sending messages of eight bytes. All data was collected from ARCHER[32].

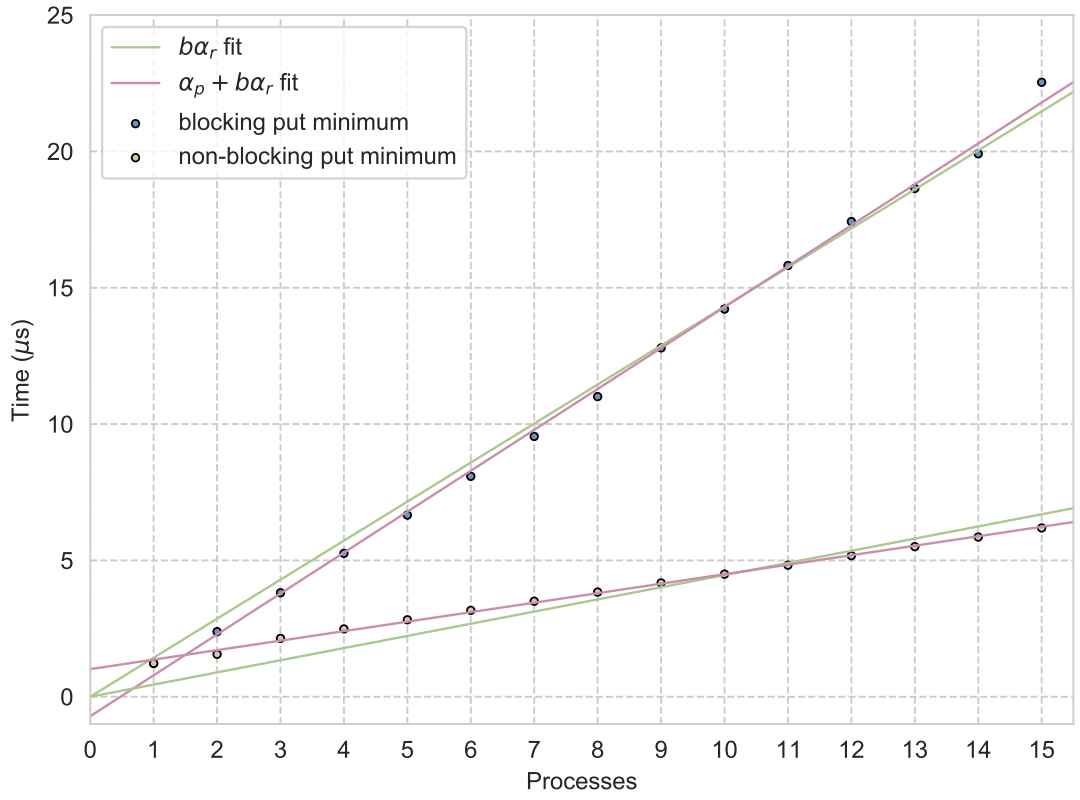


Figure 4.7: The plot shows the minimums of the timing data presented in Figure 4.6. The regressions shown fit the latency-bandwidth and pipelining latency-bandwidth model to the given data. All data was collected from ARCHER[32].

When data is transmitted through a one-sided operation, either a flag must be set on the receiving side, or the original value in the given memory location must be a sentinel value which can be identified as an *unfinished operation*. In Figure 4.6 we show a multicast operation with both blocking (*dmapp_put_flag*) and non-blocking (*dmapp_put_flag_nbi*). The non-blocking variant allows for implicit local synchronization to finalize all operations compared to the individual synchronization with the blocking version.

The motivation for the pipelining latency-bandwidth model comes from the observed data presented in Figure 4.6. The core observation from Figure 4.6 is that we are able to perform a pipelined message send, in effect constructing a multicast, which consumes less time than is required to send an equivalent number of messages using the blocking put operation.

Figure 4.6 plots the blocking and non-blocking put operations of eight bytes to a number of processes as a set of boxplots of raw data. Figure 4.7 shows regression lines overlaid on the minimum values for both the latency-bandwidth model and the pipelining latency-bandwidth model. The values of the α_p, α_r, and their ratio for the non-blocking case are presented in Table 4.1 by the first row with the message size being eight bytes. It should be noted, the absolute values of α_p and α_r are not relevant, only their ratio, since they do not represent a specific hardware feature. In other words, they only have meaning within the mathematical model of the message operation.

Figure 4.7 uses two regression lines to show effectiveness of the pipelining latency-bandwidth model. The pipelining latency-bandwidth model is also applied to the blocking data, which is a purely mathematical interpretation. The fit values are $\alpha_p = 1.50$ and $\alpha_r = -0.72$. Both of these values must be positive, therefore the negative α_r value is not physical. The non-physical aspect of the α_r value provides no further understanding of the underlying mechanisms.

The original latency-bandwidth model is unable to represent this behaviour. This can be seen by the regressions presented of the $b\alpha_r$ fit using the minimums of both sets of data. As shown the regression does not fit the non-blocking put timing data well, however when Equation 4.6 function is used the non-blocking timing data can be approximated well.

$$T_{\text{LBP\_MULTI}} = \alpha_p + b\alpha_r \tag{4.6}$$

4.3.1 Validation

The shortcomings of the pipelining latency-bandwidth model, and in effect all models, is clear from Figure 4.6. Real world machines have noise which causes a distribution for each parameter. Typically the minimum is used as a representative theoretical value since it is the closest empirical value to the theoretical models.

If we want to account for the observed noise of the measurements, more complex and fine-grained models would need to be used, such as the LoP model discussed in Section 4.2.4. Another approach to noise from a machine is to use a stochastic model which takes into account the distributions of parameters.

4.3.2 Larger Message Sizes

We purposefully disregard message size for the pipelining-latency model and therefore arrive at the model presented in Section 4.3. A cursory exploration of larger messages sizes is presented in this section. We show the declining ability of the underlying hardware to pipeline messages and therefore the lack of need for a pipelining model for large messages.

Figure 4.8 presents data collected in the same manner as in Figure 4.6. We measure each multicast operation individually using the *dmapp_put_flag_nbi* API and find the minimum time for the collected timings. This is done for each of the message sizes shown. The element size is set as a quad-word, eight bytes. The minimum timings for each message size are used to fit Equation 4.6. For each message size dataset the timings are normalized by dividing the dataset by the $Processes = 1$ value. The unnormalized α_p, α_r and the ratio of these is shown in Table 4.1 for each message size.

Comparing the fit values in Table 4.1, it is clear that the pipelining ability is reduced as message size is increased. We observe that the ratio tends to zero with increasing message size. This indicates that the pipelining capability of the network decreases with message size.

This is likely due to some resources available in the hardware being increasingly heavily consumed as the message size increases. Examples of these resources are the available injection bandwidth, the inherent pipelined processing within the NIC or in-network buffering.

As mentioned above, it is important to note that the pipelining latency-bandwidth model does not focus on any specific resource from the hardware, but intrinsically

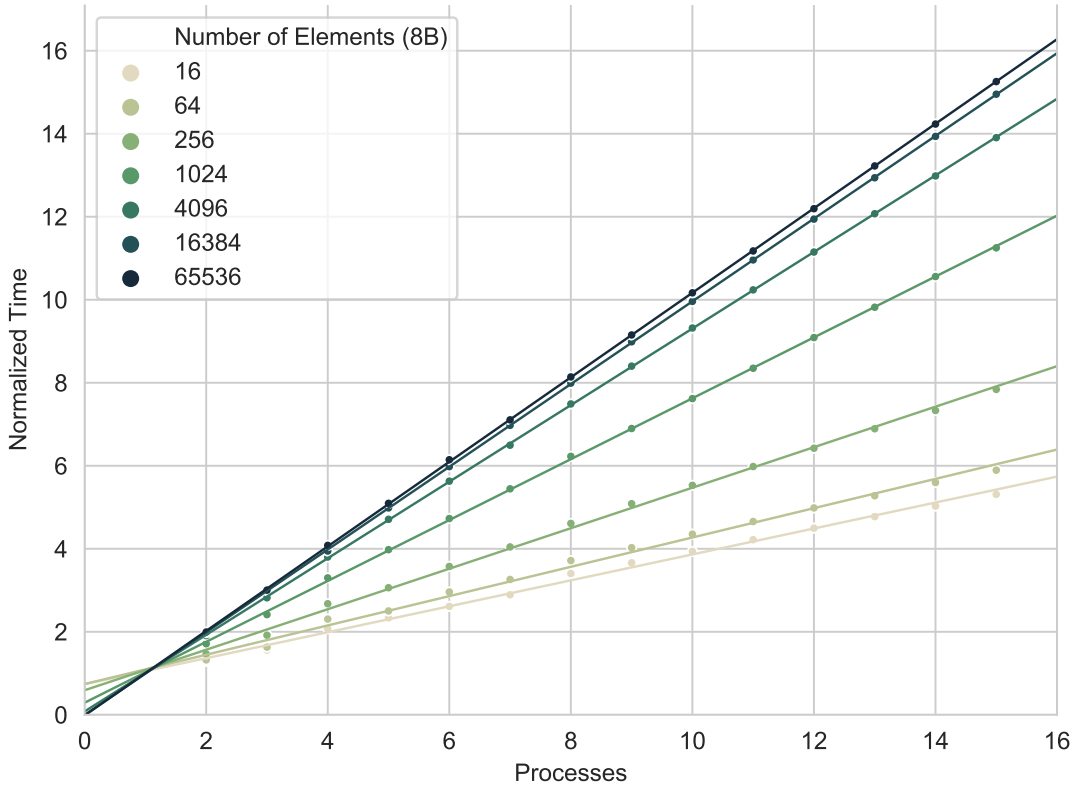


Figure 4.8: The plot shows the timing data of a sequential put operation to multiple targets with varying message size. The timing data is normalized for each message size to the $Processes = 1$. The regression lines show the difference in α_p and α_r for the message sizes.

Number of Elements	Message Size	α_p (μs)	α_r (μs)	$\frac{\alpha_p}{\alpha_r}$
1	8 B	0.88	0.38	2.3
8	64 B	0.89	0.38	2.4
16	128 B	0.91	0.38	2.4
64	512 B	0.96	0.46	2.1
256	2 KB	0.96	0.79	1.2
1024	8 KB	0.86	2.17	0.4
4096	32 KB	0.68	7.75	0.088
16384	128 KB	-0.12	30.1	-0.004
65536	512 KB	-1.7	119.3	-0.014

Table 4.1: The fitted α_p and α_r data from Figure 4.8 is shown with respect to the message size.

considers the entirety of the machine. As such, the pipelining aspect is a function of the interface to the hardware instead of a specific aspect of it. This means that the α_p and α_r parameters, fit in Table 4.1, do not carry a hardware specific meaning only an abstract meaning with a unit of time. The parameters fit the provided interface not the provided hardware given our definition in Section 4.3.

One would expect the round trip time to increase as the message size increases, but this is not the case given the values presented in Table 4.1. We suspect the non-blocking put operation only returns control to the host process after the last packet has been sent. This means that the α_p value represents the time between the last packet being sent and the last returned acknowledgement message being received. This value does not depend on the message size. However, this understanding of the mechanism causes α_r to grow with the message size.

In summary, the above finding indicates the inability of the pipelining latency-bandwidth model to represent large message pipelining. The model would need to be modified to enable the machine parameters to vary with large messages and reinforces the notion of modelling the interface and not the underlying hardware.

4.3.3 Model Comparison

In this section we compare the pipelining latency-bandwidth model to the LoP, LogfP, and LogP models to evaluate usability and accuracy. The LogP family of models is considered as the state-of-the-art modelling of networks and therefore is most modified

to capture various effects as shown in surveys[78, 120]. We show we capture all effects with a simpler set of parameters in the pipelining latency-bandwidth model.

Both the LoP and LogfP models attempt to capture the known hardware pipelining inside the NIC which occurs for small messages. The initial model, LoP, captures this accurately, but is too complex to be used as a tool for algorithm design. The model consists of six parameters which are fit to the experimental data, but several of the parameters do not have a physical meaning. The LogfP model is introduced to represent the same effect accurately, but create a suitably simple model to work with analytically. As Hoefler et al. [57] stated, "Thus, the [LoP] model is practically unusable.". Therefore, we dismiss the LoP model from the comparison since it is superseded by the LogfP model.

Figure 4.9 shows a plot from Hoefler et al. [57] showing their finding of a pipelining, processing, and saturation of the network component on an Infiniband cluster. As seen, the experimental data follows closely the LogfP prediction. The LogP model prediction is shown to be not applicable on the given hardware. Figure 4.10 and Figure 4.11 show the equivalent metric of total round trip time per process for the ARCHER and *tds* platform.

The end round trip time is taken as the reception of the acknowledge message which is returned to the sending process from the receiving NIC such that the buffer can be reused. These figures are using the same experimental data from Figure 4.7 for ARCHER. We can see that the LogfP prediction is not accurate to the modern hardware experiment. The LogfP prediction includes a global minimum of round trip time (shown in Figure 4.9 near eight processes) which does not occur in Figure 4.10 or Figure 4.11. Given this finding we dismiss the LogfP model, because it captures an effect which we did not find on ARCHER.

Due to the above finding we can equate terms in the LogP and pipelining bandwidth model as follows:

$$\alpha_r = o$$

$$\alpha_p = 2L + o$$

The gap parameter, g, is not included in the expressions above, because the LogP model allows disregarding the parameter in situations in which it is not observed. The reasoning for this is that the inverse of the per-peer communication bandwidth, inter-

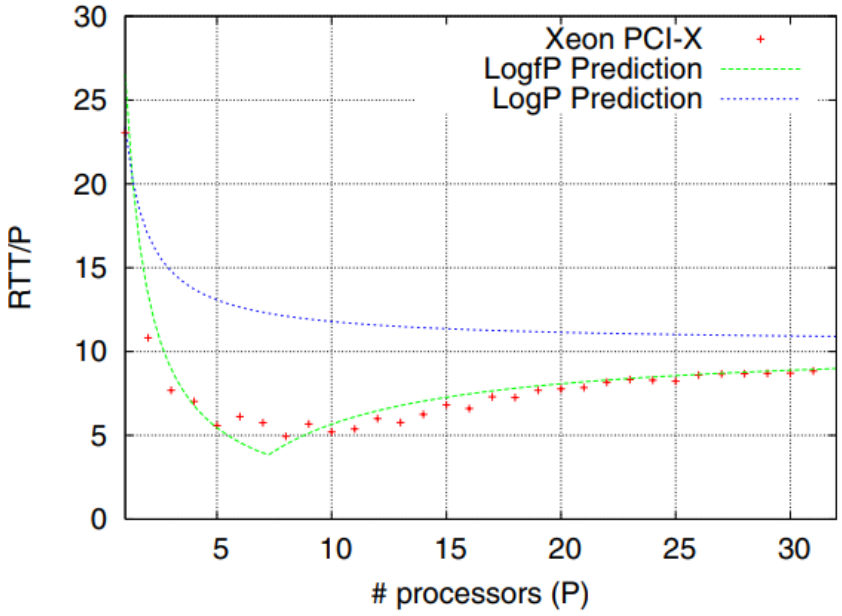


Figure 4.9: The plot shows the round trip time per process for the LogfP model on an Infiniband cluster, in addition to the LogP model prediction. The plot is copied from Hoefler et al. [57].

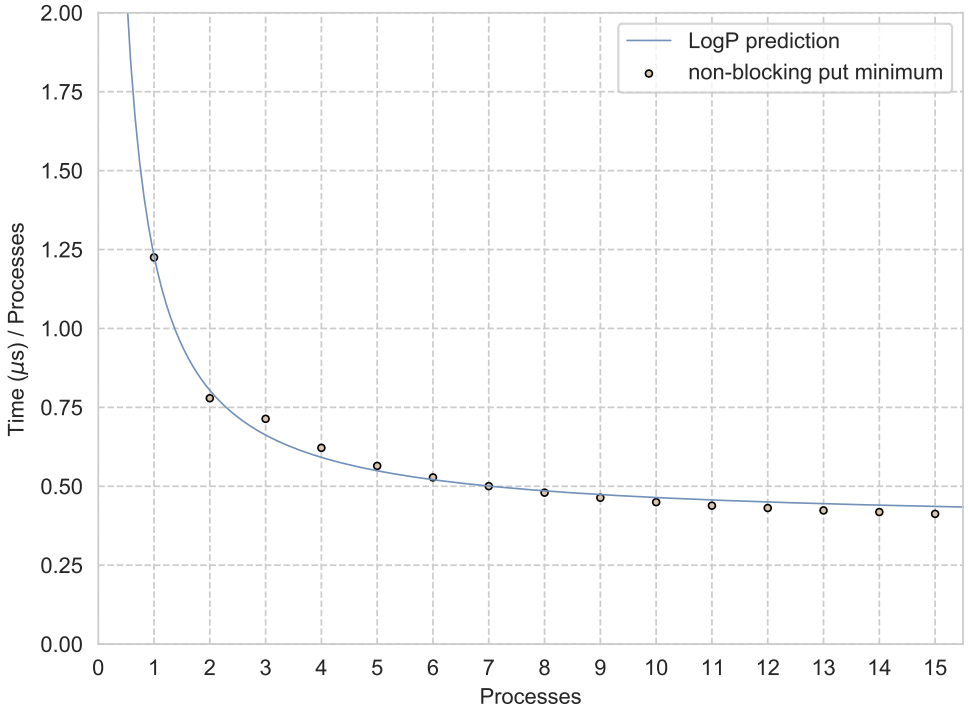


Figure 4.10: The plot shows the round trip time per remote process for the ARCHER platform.

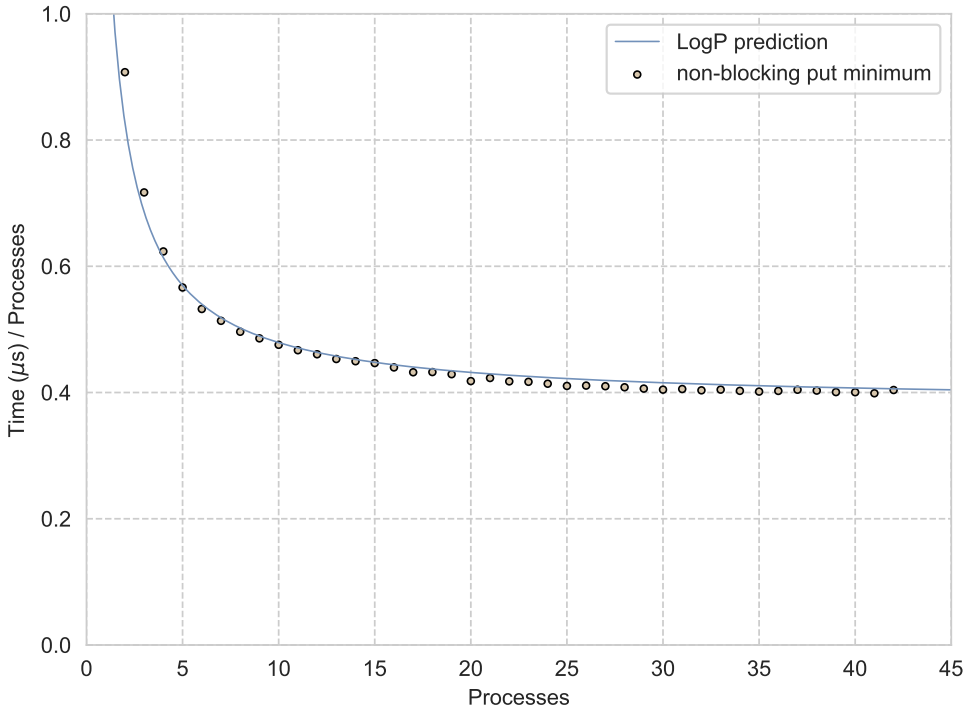


Figure 4.11: The plot shows the round trip time per remote process for the *tds* platform.

preted as the injection bandwidth in modern terms, is significantly below the overhead of issuing a message sending. The value for the injection bandwidth given in Alverson et al. [3] is 10.2 GB/s this results in $g = 6$ns which is well below the α_r value of 380ns. In addition, we set the b parameter of the pipelining latency-bandwidth model to one, since the LogP model intrinsically handles a single message.

Through the construction of the pipelining latency-bandwidth model we show an equivalence to the LogP model with intrinsic support for multicast with the b parameter (the number of messages to send). In addition, our model uses only locally knowable quantities such as the local overhead (α_r) and the round trip time (α_p). The round trip time encapsulates all underlying effects of the hardware instead of assuming the underlying hardware mechanisms.

However, the pipelining latency-bandwidth model has the disadvantage that all information of underlying processes is lost. From the model parameters the abstract multicast operation is well defined, but what steps needs to be executed in hardware is opaque. Additionally, changes to the software interface necessitate changes to the model entirely, but it is not obvious how the software interface would change in future. Finally, the pipelining latency-bandwidth model does not give a global time of arrival on the receiver-side of the communication. The conservative estimate of $T_{\text{LBP\_R}}$

(Equation 4.5) is used to ensure arrival and the safe modification of the send buffer.

4.4 Fennel Model Simulator

The *Fennel* simulator[99] was introduced to analyse algorithms in a more fine-grained approach, compared to mathematically modelling. The intention of the simulator is to bridge the gap between our theoretical understanding and real-world results. The design of the simulator is heavily based on that of LogGOPSim[59] which is further discussed in Section 4.4.1.

For this purpose the simulator had to be simple, flexible and moderately performant. However, the performance was not the priority, since only short executions of the algorithms would be simulated, not entire simulations of applications. The simulator is written using modern Python[117] due to the language's flexibility and ease of prototyping. In addition, the simulator had to be structured in such a way to support many parallel computational models and applications.

Many simulators exist in literature some of which will be discussed in Section 4.4.1. While these simulators could be used for our purposes they would require significant modification or significant amount of effort in modelling to use in the fashion we intended. This additional effort required in combination with not fulfilling all requirements directly convinced us to implement *Fennel*.

The primary reason we did not use LogGOPSim directly, which *Fennel* is directly based on, was that the simulator is written in the C language and therefore is time consuming to modify. In addition, the language does not provide object oriented facilities which are critical to a simple and understandable application. We sacrifice the performance which was targeted by LogGOPSim for the ease of modification.

The *Fennel* simulator provides a theoretical (based in reality) approach to evaluate communication patterns. With the simulator we are able to analyse aspects of an algorithm which cannot be captured with a purely mathematical model, because these do not treat processes individually. In addition, simulation allows us to experiment with modifications to models, such as machine noise, which is not easily done with a mathematical model and requires a model for the noise itself. Finally, we are able to draw simulations easily and are able to illustrate important effects visually.

The *Fennel* simulator is available as open source software[1], available to be extended. The source code consists of approximately 3000 lines of code, including tests

[1] https://github.com/martinruefenacht/fennel

and documentation. The simulator itself is not very complicated, being based on the LogGOPSim structure and priority queue implementation. However, the complexity is introduced by the visualization and program generation packages. The overall design of *Fennel*, aiming to support many different aspects of machines and algorithms, also adds complexity. This is further illustrated in Section 4.4.3.

4.4.1 Background

Several different approaches have been used to simulate the characteristics of computer networks. On the analytic end of the spectrum is queuing theory which can be used to express overall dynamics of a computer network in mathematical terms. However simulators used in HPC are developed using a more practical approach using time-stepping methods or discrete event simulation, because networks being simulated are typically too complicated and cannot be easily represented by a closed form expression.

With *Fennel* we seek to simulate many algorithms through a variety of machine models and thereby gain an understanding of the characteristics of the specific algorithm, not the model of the machine. This leads to insights which are not able to be gained with purely analytic methods. In addition, it allows us to use *Fennel* as a research vehicle which is allowed to diverge from reality.

Many simulators have been developed in the past to address a variety of requirements. One approach of such simulators[4, 51, 121] is to fully emulate an execution environment, such that the application does not know that the execution is happening virtually. Another approach in literature has been to use trace-based execution which allows a recorded trace to be executed with a simulator[18, 97, 101, 112]. The benefit of this approach is that the trace represents exactly what the application is doing without the overhead of virtual computation.

In addition to the method of simulation, time-stepping or discrete events, simulators also capture different levels of abstraction. Some simulators choose to be *cycle accurate*, which means that the hardware is simulated entirely from the CPU. While other simulators use models at various levels of abstract to represent individual components. The accuracy of the models to reality determines the predictive capability of reality by the simulator. These components can also be linked together in a variety of ways to construct simulators which attempt to represent a CPU as accurately as possible while representing network communications with an abstract model.

At first glance cycle-accurate simulators are ideal since they would capture all ef-

fects one would observe in reality, but the complexity and computational cost quickly becomes unsustainable for the HPC environment. To simulate a full size supercomputer a larger supercomputer would be required, therefore cycle accurate simulators are typically used for cases such as CPU design. Network simulations on the other hand are often performed using a discrete event simulation with an underlying cost model.

4.4.1.1 ROSS

The Rensselaer's Optimistic Simulation System (ROSS)[86] is a general purpose discrete event simulator which allows modelling of any system not just supercomputers. The simulator uses logical processes to represent state of the modelled system while events are used to affect the state. Multiple events can be used in a causally linked way to affect multiple logical processes.

The goal of ROSS is to provide a simulation system which can be run on distributed machines. This is done by handling the temporally forward event handling on multiple processes and handling a situation in which an overtaking event occurs by *unrolling* the simulation instead of check pointing. This necessitates the model developer to implement both forward and backward event handling. Given ROSS is a general purpose simulator anything can be modelled, but it also does not provide specific aids for modelling computers of any form. Therefore the amount of work required by the model developer is quite large compared to using a task specific simulator.

4.4.1.2 Parsim

The message PAssing computeR SIMulator (Parsim)[107, 108] attempts to simulate a message passing multiprocessor to enable prediction of algorithm performance on new platforms. Parsim is a task specific distributed memory computer simulator which cannot be used for any other simulations. It supports modelling of topologies through the forced InterConnection Network component. Hosts are modelled through a state machine per host.

The InterConnection Network uses a non-overlapping latency-bandwidth model for the communication cost. The computational aspect of the processing is represented a constant time per data unit per process.

Semantically Parsim is similar to *Fennel*, it simulates the computation and communication using abstract models from an abstract description of the algorithm. However

it does not explicitly use a discrete event simulation, but uses phases of computational and communication similar to an analytic model.

4.4.1.3 SST

The Structural Simulation Toolkit (SST)[87, 96] was developed to address the need to explore novel systems which interact with modern programming models. It is intended to be an open framework which uses both time-stepping and discrete event simulation on a per component basis to enable various levels of abstraction. The key contribution of SST is being able to explore the entire *machine*, the combination of microarchitecture and network, in combination with a programming model.

Conceptually SST is separated into the *frontend*, which handles the program and instruction interpretation and the *backend*, which handles the instruction and microarchitectural timing. The *frontend* supports both program traces and emulating compiled executables. The goal is to be able to swap various components transparently to other layers to allow for a variety of abstraction levels inside the simulation.

A key motivation to SST is to explore the usability of different programming models. This is done by exposing *backend* features such as multithreading or offloading to the *frontend* to be used by a program. In addition, this allows MPI or OpenMP to be supported directly without abstraction.

4.4.1.4 LogGOPSim

The LogGOPSim simulator[59] was developed to study parallel applications and algorithm behaviour in various network and system models. The simulator implements the LogGOPS model, which they introduced, as an extension of the LogGPS model. It implements a trace-based ingestion of the program which is used to capture the behaviour of large scale applications.

Fennel is largely based on the contributions which LogGOPSim made. The architecture of the simulator is similar and the discrete event simulation is performed using the same approach with a priority queue for temporal ordering of events. Unlike *Fennel* LogGOPSim also implements the MPI point-to-point semantics using a unexpected message queue and receive queue, which aids simulating applications. In addition, later versions include a network simulation which is intended to allow for experimentation with congestion in the network.

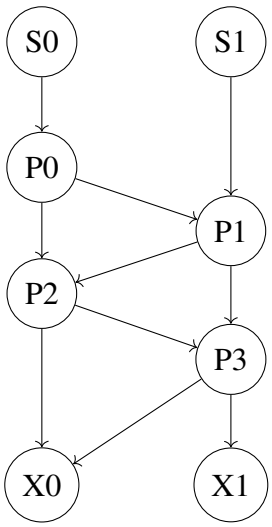


Figure 4.12: PingPong Program Representation

4.4.2 Program Representation

Applications are represented abstractly to capture the communication and computation dependencies within the *Fennel* simulator. This achieves the desired goal of simulating the overall communication behaviour and not to be overly complicated and therefore performance intensive.

The applications are encoded as a Directed Acyclic Graph (DAG) which at each node has a task associated as metadata. This allows the model implementation of the machine to walk the graph and thereby execute the control flow. The tasks are generic operations which are provided on typical hardware. In addition, helper tasks are implemented which allow easier construction of the DAGs by automated means, for example initialization tasks and proxy tasks.

Figure 4.12 shows the DAG for a PingPong application and Figure 4.13 shows an AllReduce operation. The task at each node contains the parameters of each task and the node on which it is supposed to be executed. Tasks are scheduled for execution once all dependencies are fulfilled. The task types are as follows:

StartTask The StartTask task type is a helper task for the simulator to easily find entrant nodes in the program graph. This also allows for simple representation and segmentation between separate executions within the same model instance. The task type when encountered in the simulator does not require any simulation time to execute.

ProxyTask The ProxyTask task type is similar to the StartTask type, because it does

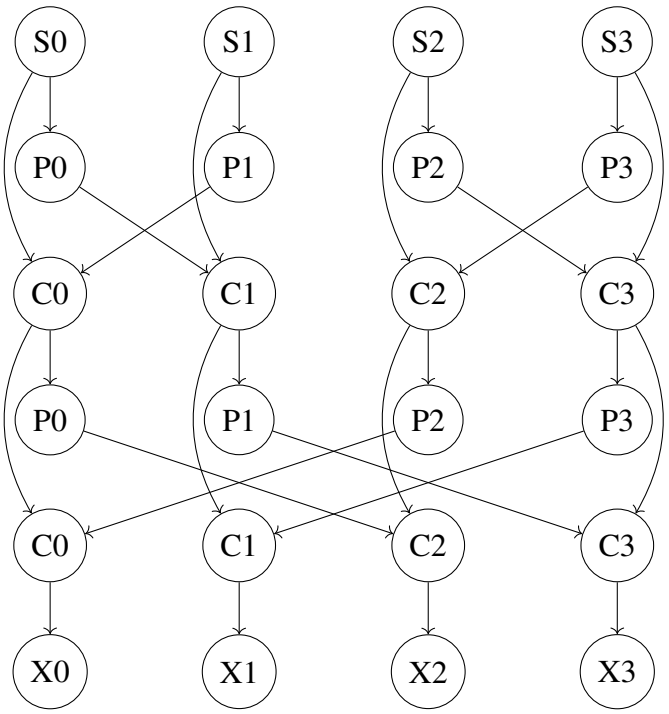


Figure 4.13: Recursive doubling algorithm AllReduce program representation.

not require any simulation time to execute. It acts as a link between actual simulation task types. This allows for simplified algorithmic generation of the schedules through the program generator.

SleepTask The SleepTask is provided as a way to block a process in the model from executing until a certain time is elapsed. This can be used to probe skew in program execution.

ComputeTask The ComputeTask task type is given to simulate a given local computation, such as a compute phase in an application. The ComputeTask is the same as the SleepTask since it blocks the process for a certain amount of time, however it is helpful to be able to label tasks.

PutTask The PutTask task type is the most important task as it describes the abstract operation of a put communication between processes. The specific network operations which can be encapsulated in this task are defined and implemented by the given model.

GetTask The GetTask task type is an encoding of a *Get* communication operation. As with the PutTask, the GetTask is implemented by the model.

In addition to the task information, each vertex of the DAG also contains a *concurrency* flag and an *any* integer parameter. By default, all vertex dependencies need to be fulfilled in order for a task to be scheduled for execution. However, the *any* parameter allows the execution to be triggered with only a subset of the incoming edges being fulfilled.

Creation of these application DAGs can be performed in multiple ways. The method chosen by LogGOPSim is to generate a trace of an MPI application and then using the GOAL[60] language to encode the communication pattern. We opted to generate the DAGs from the intended algorithm. To this end multiple generator functions have been implemented in order to replicate the communication and computational patterns for important algorithms.

Both the LogGOPSim and *Fennel* can use DAG programs generated by either the trace or the generation method. With *Fennel* we chose the direction generation from algorithms due to the interest in algorithmic effects instead of application level effects. Using a generation method from an analytic algorithm enables use of simple message structures, e.g. uniform size. Application traces would typically be more complex with varying message sizes, while these could be represented in *Fennel* this was left as a future task. In addition, application traces are less constrained than analytically generated programs.

4.4.3 Simulator Architecture

The structure of the *Fennel* simulator is shown in Figure 4.14. Compared to other simulators which are purposed to simulate entire MPI applications and therefore need to focus on performance, our architecture reflects the goal of flexibility and exploration.

The *Machine* object is the core of the *Fennel* simulator. A discrete event simulation engine is implemented using the next-event time progression. In our case, the event progression mechanism is implemented using a priority queue. With this choice of architecture, the assumption is made that the system does not change except when an event happens.

In addition to the event engine, the *Machine* object is composed of a compute model and a network model. This architecture is beneficial compared to a direct inheritance, because it allows flexibility without having to re-implement a combinatorially large number of models which serve a near identical purpose. This design ensures, with respect to models, the class explosion problem is avoided. The compute model

and network model are interfaces which facilitate the processing of their specific task types.

In addition to the models, the *Machine* contains instruments which serve as a method to capture information which is otherwise lost during the *run-time* of the *Machine*. The instruments are implemented using the observer pattern: they are registered for events which occur during program execution.

4.4.4 Capabilities

The *Fennel* simulator is able to execute any program which can be constructed from the given tasks in Section 4.4.2. The simulator is capable of executing programs in two modes. The first is drawing the simulation as seen in Figure 4.15 and Figure 4.16, which are respectively the execution of the programs shown in Figure 4.12 and Figure 4.13. Complex models with stochastic effects can be drawn, but due to their nature they evaluate to different results during each execution, which makes drawing less useful except for illustrative purposes.

The second mode of the *Fennel* simulator is the measurement mode. This mode is intended to be used with complex models which include stochastic effects, such as noisy networks, and therefore go beyond simple mathematical models. The instruments, which can be registered with a machine, become useful in this instance to evaluate the properties of the execution.

As shown in Figure 4.14 five models are currently implemented which are either implementations of the compute or network model. In addition, *noisy* versions of these exist through inheritance and allow for stochastic measurements. The stochastic nature of these models will be further discussed in Section 4.4.5. The goal of the *Fennel* simulator was to implement a simulator which does not replicate experimental results, but allows exploration of algorithms on a variety of models, which match reality to varying degrees.

Currently two instruments are implemented: the first is a recorder instrument which records all timing information about tasks, and the second is the bandwidth instrument which serves as a bandwidth measurement tool. The bandwidth is determined by a model: however, when a non-deterministic model is used an experiment bandwidth value can be derived.

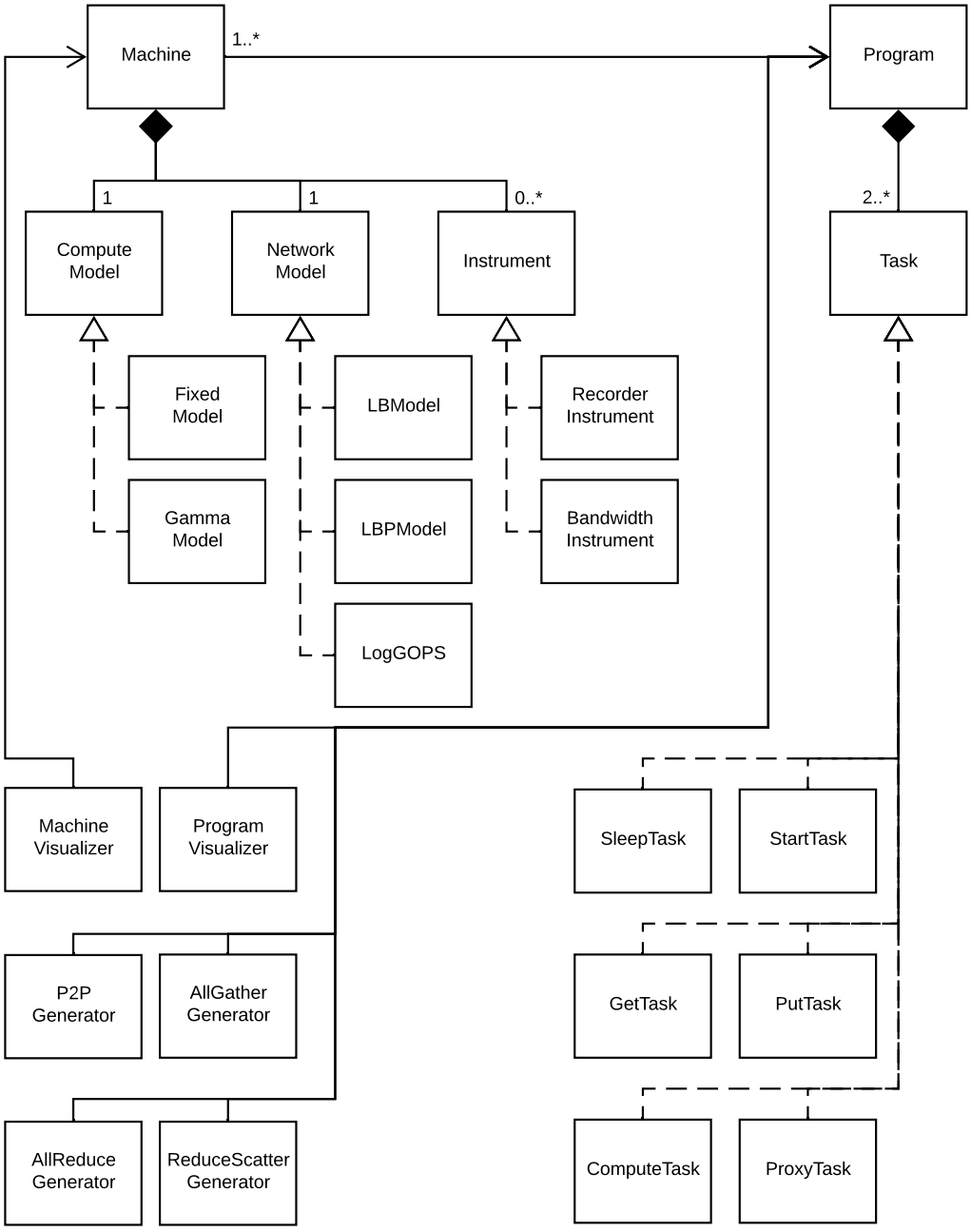


Figure 4.14: Fennel Simulator UML

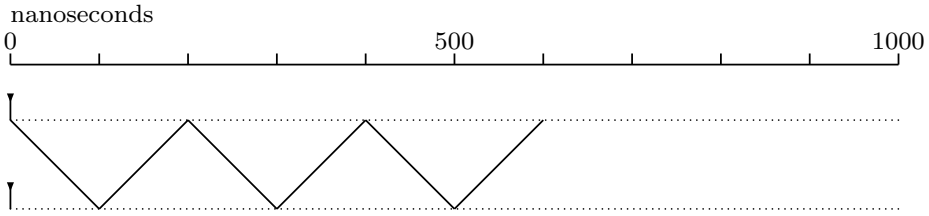


Figure 4.15: PingPong simulation

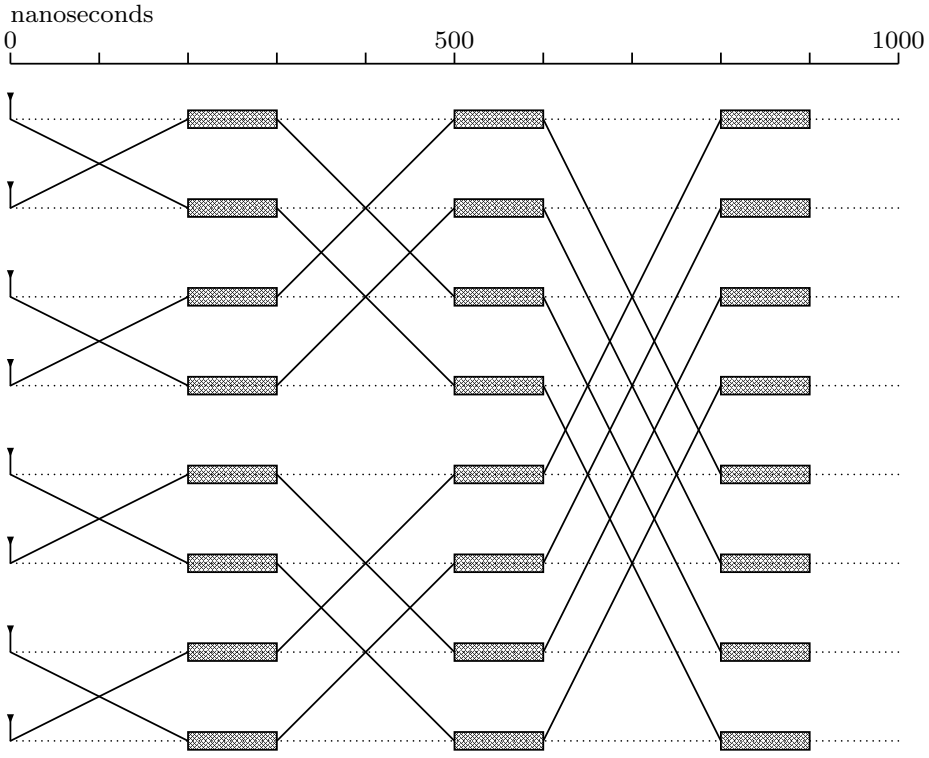


Figure 4.16: recursive doubling simulation

4.4.5 Validation

Validation of the simulator has to be done using a variety of testing methods. The models themselves need to be verified and tuned to represent the underlying physical machine, as shown in Section 4.3. This implies the model needs to be able to represent reality. In other words simple models cannot represent reality due to the innate lack of variables, but more complex stochastic models could.

The pipelining latency-bandwidth model uses direct measurements (using the minimum found) for the variables which allows absolute certainty that the model accurately represents the given interface, but factors such as the probability distributions of those variables is not currently accounted for, as seen in Figure 4.6 and Figure 4.7.

Validation of the general simulation is done through a test suite which contains both unit tests and integration tests. The unit tests verify the components, such as the task implementations or the program representation. Complex entities, such as the core *Machine* object, are verified using the integration level testing.

Integration tests are performed by generating programs and executing these programs with known deterministic models. This verifies that the common processing which takes place for all models correctly executes and is consistent and correct. This is done by measuring the total execution time after executing a program on a machine and verifying by comparing to an analytically calculated reference value. The integration tests include common patterns such as ping-pong, multicast, and recursive doubling. In addition, the visual output is verified by comparing to manually verified visualizations.

Chapter 5

Recursive Multiplying

The performance of AllReduce is crucial at scale. The current defacto AllReduce algorithm, recursive doubling with pairwise exchange, theoretically achieves $O(\log_2 N)$ scaling for short messages with N peers, but is limited by improvements in network latency. A multi-way exchange can be implemented using message pipelining, which is easier to improve than latency. Using our method, recursive multiplying, we show reductions in execution time of between 8% and 40% for AllReduce on a Cray XC30 over recursive doubling. Using a custom simulator we further explore the dynamics of recursive multiplying.

5.1 Introduction

5.1.1 Overview

This chapter discusses our implementation of the AllReduce operation. It contains a discussion of the requirements by Message Passing Interface (MPI) for AllReduce operations and some prior algorithms used to implement the operation in Section 5.2. A discussion of directly related methods is given in Section 5.3.

We introduce the *recursive multiplying* algorithm in Section 5.4 with an analytical derivation from the pipelining latency-bandwidth model in Section 5.4.1. The implementation of recursive multiplying is presented in Section 5.4.3. In Section 5.4.4, we introduce a heuristic method by which a suitable schedule can be found.

Experimental results are presented in Section 5.5 and simulations using the *Fennel* simulator are presented in Section 5.6.

5.1.2 Contributions

This chapter makes the following contributions:

- A generalisation of recursive doubling, *recursive multiplying*, is introduced to implement MPI_Allreduce.
- It is shown that the *recursive multiplying* method allows for lower latency than previously seen for AllReduce.
- The prime merging method is introduced to handle non-power-of-two process counts, which supersedes the collapse and expand method in most situations.
- A heuristic method is given to determine a near optimal schedule for an AllReduce operation using *recursive multiplying* and prime merging.

The work presented in this chapter is based on the published conference article[98] and the extended journal version of the article [99]. In addition to the two articles, the chapter contains evaluations and explorations which have not been published.

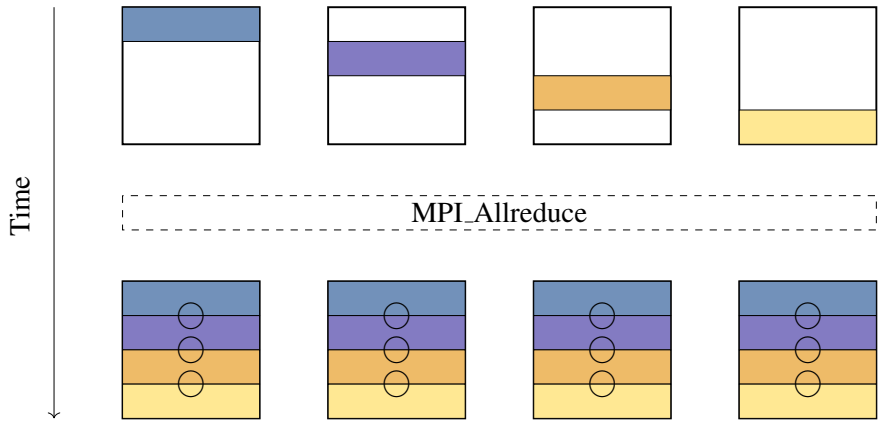


Figure 5.1: An illustration of an MPI_Allreduce operation which shows the ordering requirements and contributions of each participating process. The order shown in the resulting processes illustrates the order of operations.

5.2 Background

5.2.1 MPI_Allreduce Definition

The MPI specification defines the MPI_Allreduce interface which performs the AllReduce operation across a communicator with a specific operator. Importantly, the specification requires identical results on all participating processes, *spatial consistency*, and an *advice to implementors* note suggests consistency for when the function is called multiple times with the same arguments, *temporal consistency*. Ignoring the temporal consistency when implementing or designing an algorithm for MPI_Allreduce could result in an extremely difficult to diagnose anomalous behaviour. The MPI Standard determines the *canonical* order of operations by the ranks of the processes in the group[85].

Figure 5.1 illustrates the MPI_Allreduce functionality. Each colored portion of data needs to be appropriately exchanged in order to achieve the AllReduce operation.

The operation performed when combining intermediary results is assumed by MPI to always be associative. User defined operations are able to be declared to not be commutative, but MPI standard operations are also assumed to be commutative. Due to this, the order of operations is important when using a user defined operation. For example, integer arithmetic is both associative and commutative, while floating point operations are commutative, but they are not associative. Within MPI, floating point non-associativity is acknowledged to contradict the assumption of associativity, but is accepted as an allowable optimisation for implementations. A user defined operation

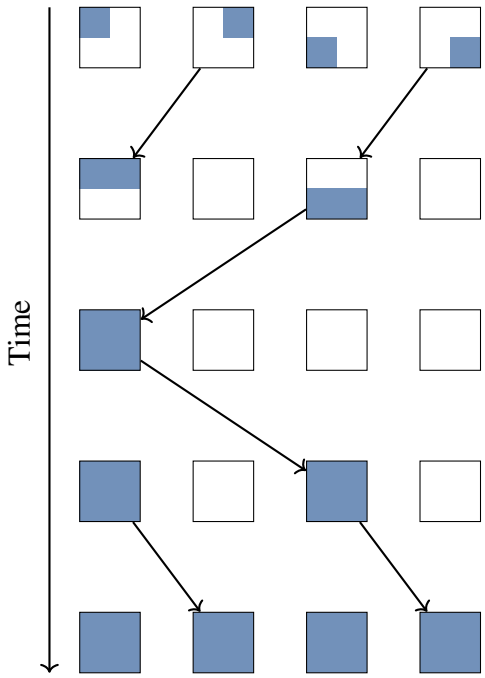


Figure 5.2: Illustration of an AllReduce operation utilizing binomial tree for a reduce and a broadcast.

such as matrix multiplication is associative, but would be labelled as non-commutative and therefore needs special treatment.

5.2.2 Algorithms

This section will discuss algorithms present in literature which fulfil the requirements set out in Section 5.2.1. This is *not* an exhaustive listing of all known algorithms. Topology-aware algorithms are specifically not discussed, because our contribution presented in Section 5.4 is not topology-aware.

Algorithms of interest are constructed from point-to-point operations and are used as such within MPI implementations. Early algorithms are basic patterns which are used for many implementations of operations: these typically are used to implement low latency collective operations.

To achieve low bandwidth usage, simple patterns are combined to construct composite algorithms. The choice between which algorithm is used for a specific instance of a collective operation is usually determined by the message size of the collective.

5.2.2.1 Fan-In/Fan-Out

A common approach to implement AllReduce in early versions[6, 110] of MPI libraries was to use a fan-in/fan-out pattern. This consists of reducing the vector to the root, typically the zeroth rank, using a minimum spanning tree and then broadcasting the resulting reduced vector to all ranks, using the same spanning tree.

Figure 5.2 illustrates the communication pattern using binomial trees. This method of implementing an AllReduce operation is not efficient, because both the computational units and communication network are underutilized. For Figure 5.2 and further diagrams the order of operations is assumed to be correct. The visualizations of the processes are partially filled to illustrate partial reductions and colors are used to show process origin when larger vectors are involved.

The complexity of this method using the standard latency-bandwidth model, discussed in Chapter 4 is presented in Equation 5.1. The process count is given as N. If N is not a power of two then the tree used must be adjusted for the specific value in order to perform the correct communication pattern considering consistency discussed in Section 5.2.1. The n parameter is the number of bytes to be reduced per process.

$$T_{\text{TREE}} = 2\lceil \log_2 N \rceil (\alpha + n\beta + n\gamma) \tag{5.1}$$

The main disadvantage of this method is the factor of two which is always present, because the method is performing two separate tree operations sequentially. The usage of binomial trees, for a topology unaware method, allows for a minimum latency operation, but other trees such as binary or Fibonacci may also be used which support better optimization for larger messages or specific process counts.

5.2.2.2 Recursive Doubling

The recursive doubling method for AllReduce is a straightforward extension of the usage of trees[6, 20, 109]. Instead of performing a single send and then inactivating a process, the process is kept active and the peer sends its contribution. This method is illustrated in Figure 5.3. By doing this the broadcast communication is interlaced with the reduction operation and thereby prevents idle time while communicating. Recursive doubling does introduce redundant computation on all participating nodes, which depending on the machine properties may not be ideal. The method is also only applicable to process counts which are a power of two. The algorithm can also be seen as a dimension by dimension reduction over a binary hypercube.

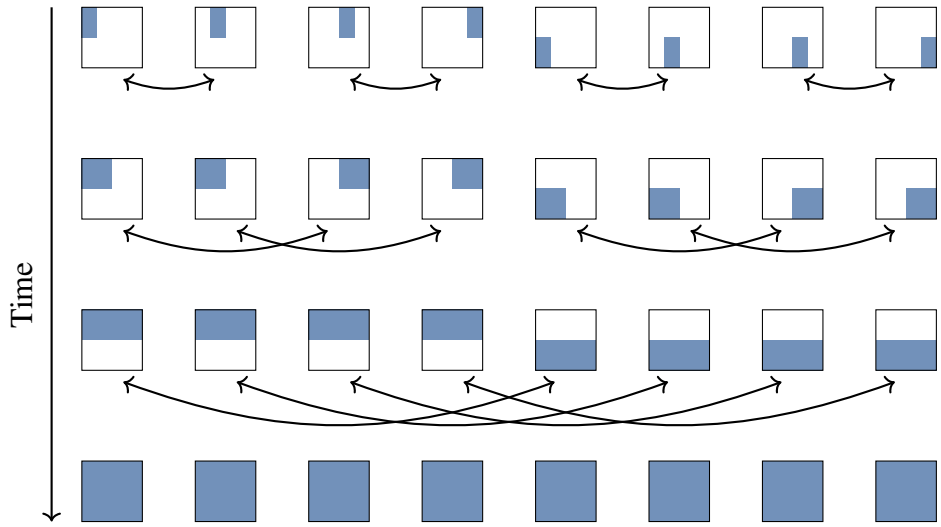


Figure 5.3: An illustration of recursive doubling for a power-of-two number of processes.

Equation 5.2 calculates the total time for the recursive doubling method using the standard latency-bandwidth model. The process count, N, is required to be a power of two. As previously, n is the vector length for each process.

$$T_{\text{RD}} = \log_2 N(\alpha + n\beta + n\gamma) \mid N = 2^k, k \in \mathbb{N}^* \tag{5.2}$$

The recursive doubling method is ideal for small process counts, because it calculates along the minimum spanning tree, which is half of the fan-in/fan-out method. While the method is applicable to a smaller number of process counts it outperforms the fan-in/fan-out method by a factor of two. For non-power-of-two process counts a collapse and expand method can be used to ensure a reasonable process count is used. This method adds an additional stage at the beginning and at the end of the recursive doubling stages, resulting in Equation 5.3.

$$T_{\text{RD-NP2}} = (\log_2 N + 2)(\alpha + n\beta + n\gamma) \mid N \neq 2^k, k \in \mathbb{N}^* \tag{5.3}$$

Algorithm 1 presents the pseudocode for the recursive doubling method in addition to the collapse/expand stages which may be required. The collapse/expand method will be further discussed in Section 5.4.2.

5.2.2.3 Composite Algorithms

Prior methods of implementation of the AllReduce operation have focused on minimizing the latency component of the algorithm to optimize for small vectors. However

Algorithm 1 Recursive Doubling AllReduce

```
1:  procedure ALLREDUCE(rank, size, local)
2:      global ← local                                  ▷ initialize variables
3:      mask ← 1
4:      pof2 ← 2^⌊log2(size)⌋                           ▷ nearest lower power of two
5:      rem ← N − pof2                                  ▷ remainder

6:      if rank < 2 × rem then                          ▷ collapse to power of two
7:          if rank (mod 2) = 0 then
8:              Send global to rank + 1
9:              myrank ← −1                             ▷ deactivate
10:         else
11:             Recv recvbuf from rank − 1
12:             Reduce global rbuf
13:             myrank ← rank/2                         ▷ change rank
14:     else
15:         myrank ← rank − rem
16:     if myrank ≠ −1 then                             ▷ recursive doubling
17:         while mask < pof2 do
18:             newdst ← myrank ⊕ mask                  ▷ find peer
19:             if newdst < rem then                    ▷ virtual to real rank
20:                 dst ← newdst × 2 + 1
21:             else
22:                 dst ← newdst + rem
23:             Sendrecv sendbuf rbuf from dst
24:             Reduce global rbuf
25:             mask ← mask ≪ 1                         ▷ increment stage
26:     if rank < 2 × rem then                          ▷ expand to remainder
27:         if rank (mod 2) ≠ 0 then
28:             Send global to rank − 1
29:         else
30:             Recv global from rank + 1
31:     Return global
```

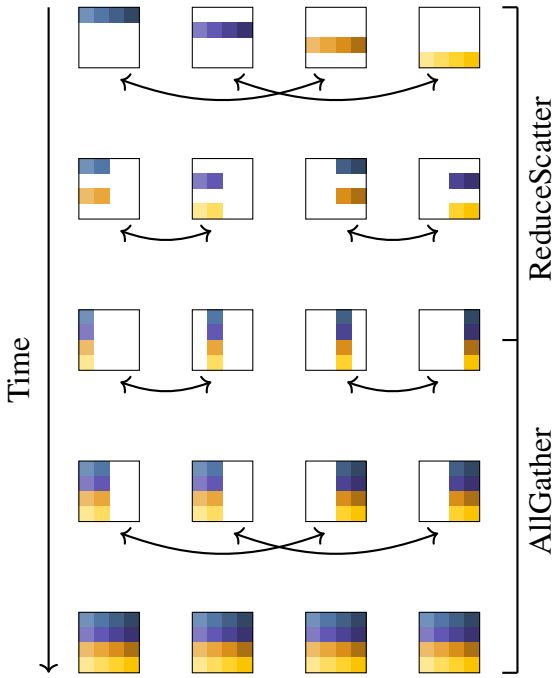


Figure 5.4: An illustration of a composite AllReduce operation utilizing a ReduceScatter (implemented by recursive halving) and AllGather (implemented by recursive doubling) for large messages.

when performing an AllReduce operation with large vectors the latency is less important than the bandwidth requirement of the underlying method. Rabenseifner et al.[95] introduced a composite method which uses a ReduceScatter operation followed by an AllGather operation. The work is based on the bandwidth optimal broadcast introduced by Van de Geijn et al.[7, 103]. Van de Geijn et al. improved upon the minimal spanning tree broadcast for bandwidth-limited use-cases by using a combination of a Scatter followed by an AllGather collective to reduce the time needed. This method improves as more processes participate.

Figure 5.4 illustrates the composite method. The presented version uses recursive halving for the ReduceScatter and recursive doubling for the AllGather. Recursive halving is analogous to recursive doubling. The method sends half vectors instead of whole vectors to the peer process while simultaneously halving the distance between peers instead of doubling. The collective operations which make up the composite method are able to be implemented with multiple underlying methods similar to AllReduce, therefore the choice of which method to use for these phases influences the overall effectiveness.

For process counts which are a power of two the complexity using the latency-

bandwidth model of the composite method is:

$$T_{\mathrm{RS}} = 2\alpha\log_2 N + 2\frac{N-1}{N}n\beta + \frac{N-1}{N}n\gamma \tag{5.4}$$

To compare the two methods performing an AllReduce operation in the latency-bandwidth model we take the difference between the two, resulting in Equation 5.5. With Equation 5.5 we see that the latency term is independent of n and remains constant. With small n the negative latency term dominates, which means the recursive doubling method is better for small n. For large n the positive bandwidth term dominates and results in a composite method providing a lower total time-to-solution.

$$T_{\mathrm{RD}} - T_{\mathrm{RS}} = -\alpha\log_2 N + n\left(\beta\left(\log_2 N + \frac{2}{N} - 2\right) + \gamma\left(\log_2 N + \frac{1}{N} - 1\right)\right) \tag{5.5}$$

5.2.2.4 Elimination

Implementations of AllReduce typically focus on process counts which are a power-of-two. However as the size of machines increases and more processes are used the powers of two become more sparse. Therefore, future methods are required to address non-power-of-two process counts more often.

Rabenseifner et al.[95] introduced a method by which *eliminations* are used to allow for better composability of the process counts and thereby address the issue of non-power-of-two process counts. The two eliminations are the 3-2 elimination and the 2-1 elimination. The 3-2 elimination is constructed such that two processes absorb and reduce the vector of the third process. The 2-1 elimination is the same as the collapse/expand method used in recursive doubling, but instead of being used only at the beginning and end of the algorithm these eliminations can be used throughout.

The complexity achieved for the AllReduce operation is $\lceil\log_2 N\rceil + 1$ for small sized latency optimized operations of non-power-of-two process counts. Figure 5.5 presents a latency optimized AllReduce operation across seven processes using *3-2 eliminations*. The third process is eliminated using the first 3-2 elimination, after which only six processes are active. Using two groups of three the six processes are reduced and finally an expansion has to be performed to move the final result to all processes.

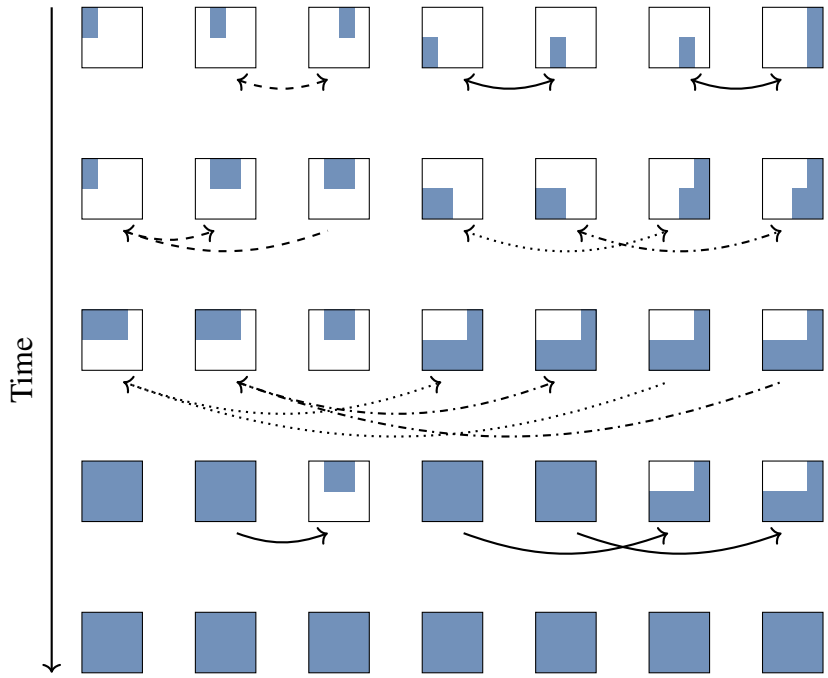


Figure 5.5: An illustration of a binary block AllReduce of seven processes. Dashed and dotted lines are used to highlight the three *3-2 eliminations* used in the collective operation.

5.3 Related work

Motivated by the LoP and LogfP model, Hoefler et al.[56] introduced a barrier operation based on the f-way dissemination pattern which allows for higher performance. The f-way dissemination pattern is an extension of the original dissemination pattern also used for a barrier operation[11, 50]. By allowing a process to send multiple messages the scaling of the dissemination barrier is improved from $O(\log_2 N)$ to $O(\log_f N)$ with the f-way dissemination barrier.

End et al.[34] introduced the f-way dissemination AllReduce which allows for a large improvement on InfiniBand. When $N \neq (f+1)k$ there is potential for duplication. An adaption is presented which performs a post process when duplication occurs, based on the data boundary from the previous stage. This allows for the correct result to be computed. Since butterfly-like patterns require an associative operator, this algorithm is only suitable for a subset of use cases.

5.4 Recursive Multiplying Algorithm

We introduce the *recursive multiplying* method as a generalisation of the recursive doubling method presented in Section 5.2.2.2. Utilizing the insight gained from the pipelining latency-bandwidth model from Chapter 4, we construct an AllReduce operation. By sending multiple redundant messages over the network, exploiting the pipelining capability, we succeed in enabling a lower latency AllReduce. Similar to other collective algorithms, we attempt to maximize the use of the network and computational facilities to perform the AllReduce in minimal time to solution.

5.4.1 Derivation

The generalisation of recursive doubling evolves directly from the pipelining latency-bandwidth model. The recursive doubling algorithm was developed during an era in which multiple messages were at best scaling linearly given the latency-bandwidth model and the present hardware at the time.

Since $\beta << \alpha$ and $\gamma << \alpha$ are assumed for the latency-bandwidth models we simplify the equations with $n = 0$. Since recursive doubling is only intended for small messages and the composite method by Rabenseifer et al.[93] is ideal for large messages, this simplification does not skew our results. Capturing recursive doubling in the pipelining latency-bandwidth model results in Equation 5.6, expressing the total time for the AllReduce operation using recursive doubling. A brief discussion of larger message sizes using *recursive multiplying* is given in Section 5.4.5.

$$T_{\mathrm{RD}} = (\alpha_p + \alpha_r)\log_2 N \mid N = 2^k,\ k \in \mathbb{N}^* \tag{5.6}$$

The pipelining model allows multiple messages to be sent for a slight increase in time compared to a single message, as demonstrated in Section 4.3.1. Using this insight, we are able to flatten the computational graph of recursive doubling by sending multiple messages containing the same information to multiple processes. We call this method *recursive multiplying* with the total time for an AllReduce given by Equation 5.7.

$$T_{\mathrm{RM}} = (\alpha_p + b\alpha_r)\log_{b+1} N \mid N = (b+1)^k,\ k \in \mathbb{N}^*,\ b \in \mathbb{N}^* \tag{5.7}$$

The b variable controls the *fan-out* of the *recursive multiplying* method: it is the number of messages to be sent by each process in a single stage, i.e. the multicast

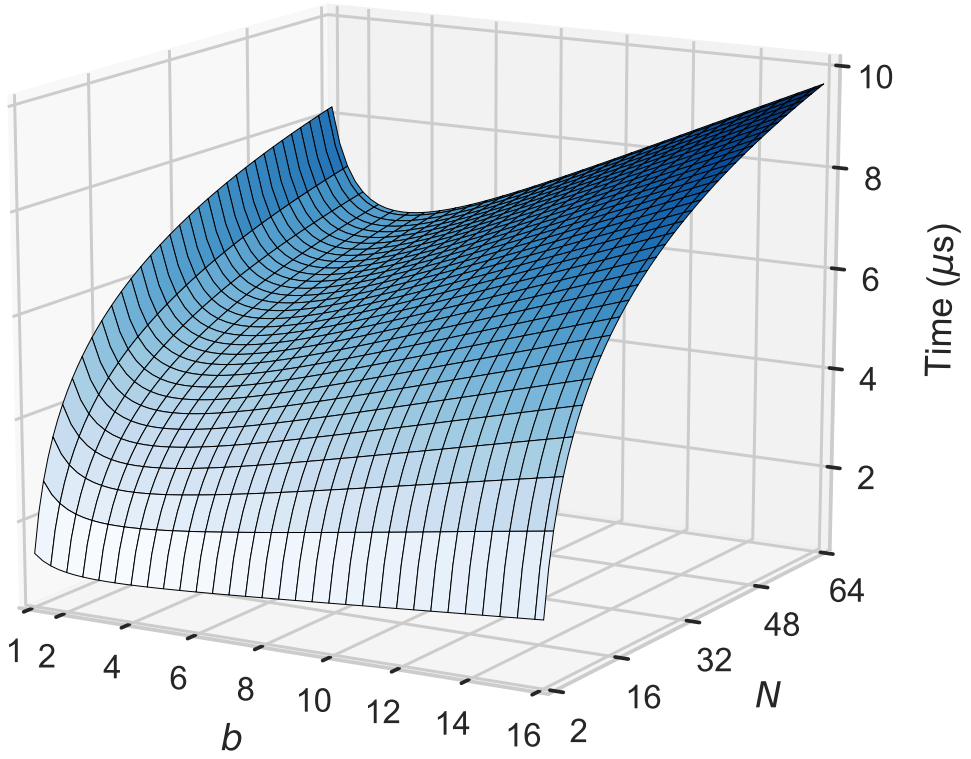


Figure 5.6: The total time for recursive multiplying given the b and N parameters with measured machine parameters from Chapter 4. This function is not evaluated in integer space.

width. In the case $b = 1$ the *recursive multiplying* method is equivalent to recursive doubling. It is important to note that both methods have a limited domain and cannot be applied to all natural numbers N: recursive doubling operates only on powers of two, while *recursive multiplying* operates on powers of $b+1$.

The b parameter of Equation 5.7 is an independent variable: it is free to be chosen by the user to form any pattern, with a range of $[1, N-1]$. An optimal value of b exists, b_{opt}, which is the value at which the overall time spent within the collective operation is minimal. With T_{RM} defined as the total time to solution of an AllReduce operation, b_{opt} minimizes the total time with respect to b. The overall program runtime is minimized if the program is network limited and not computationally limited.

Figure 5.6 shows the total time required by *recursive multiplying* given by Equation 5.7 over the given range of the b and N parameters. This figure and equation

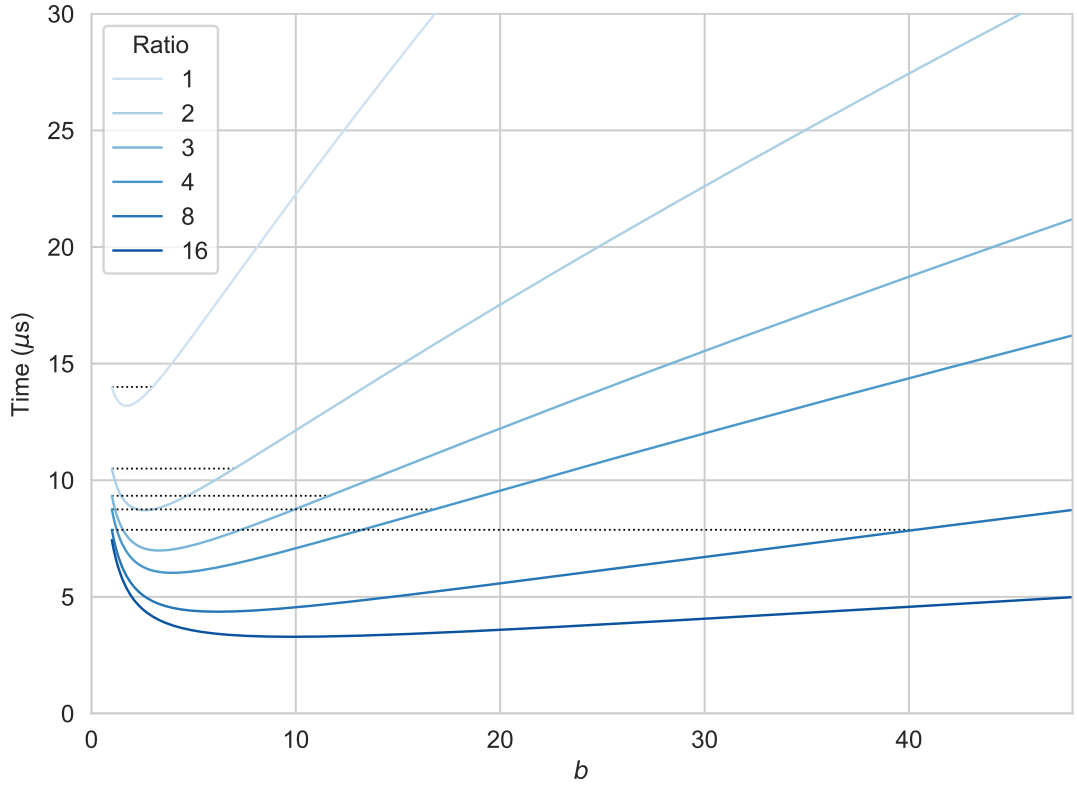


Figure 5.7: The total time for recursive multiplying with a fixed $N = 128$ with a variety of machine parameter ratios $\frac{\alpha_p}{\alpha_r}$. These functions are not evaluated in integer space.

assume N is factorisable by $b+1$. This is not true for all values of N, but is useful for the analysis, therefore we do not restrict b or N to be integers. In reality we will be required to operate within integer space. In Figure 5.6 a minimum occurs at $b \approx 3.258$ for the total time of the collective operation.

We derive an expression for b_{opt} from Equation 5.7 ignoring integer requirements. First, we find the partial derivative with respect to b:

$$
\begin{aligned}
T_{\text{RM}}(b,N) &= (\alpha_p + b\alpha_r)\log_{b+1} N \\
\frac{\partial T(b,N)}{\partial b} &= \frac{\ln N}{\ln^2(b+1)}\left(\alpha_r \ln b + 1 - \frac{b\alpha_r + \alpha_p}{b+1}\right)
\end{aligned}
$$

Second, we find the minimum of Equation 5.7 by setting the derivative to zero.

$$
\begin{aligned}
0 &= \frac{\ln N}{\ln^2(b+1)}\left(\alpha_r \ln(b+1) - \frac{b\alpha_r + \alpha_p}{b+1}\right) \\
0 &= \alpha_r \ln(b+1) - \frac{b\alpha_r + \alpha_p}{b+1} \\
\frac{\alpha_p}{\alpha_r} &= (b+1)\ln(b+1) - b
\end{aligned}
$$

Finally, we solve for b to find the minimum of Equation 5.7 given by Equation 5.8 using the Lambert W function[72].

$$
b_{\text{opt}} = e^{W_0\left(\frac{1}{e}\left(\frac{\alpha_p}{\alpha_r} - 1\right)\right)+1} - 1 \tag{5.8}
$$

Interestingly, b_{opt} depends only on the ratio of the two machine parameters, α_p and α_r, but not on the value of N. Since each stage of the collective, represented by the logarithmic term in Equation 5.7, is itself a multicast operation it follows that N does not influence the b_{opt} value. Equation 5.8 is plotted for a range of ratios of machine parameters in Figure 5.8.

Figure 5.7 presents plots of Equation 5.7 with fixed $N = 128$. This shows the increasing flatness of the curve when the ratio of the machine parameters increases. The dotted lines show the intersection of the lines from $b = 1$. The b_{upper} value is the point after which it is only effective to use $b = 1$. Therefore, values up to and including $\lfloor b_{\text{upper}} \rceil$ for the *recursive multiplying* algorithm are useful.

With Figure 5.6 and Figure 5.7, which show Equation 5.7, it is difficult to have an intuitive understanding of the combination of machine parameters and collective parameters. Equation 5.8 shows the relationship between the machine parameters and

the optimal choice of b, but only the ratio $\frac{\alpha_p}{\alpha_r}$ is important and that as this ratio increases b_{opt} also increases. From Figure 5.8 we can see the relationship of the change is not linear. In essence, a large machine parameter ratio implies greater pipelining capability, therefore a schedule generation has access to larger multicasts (i.e. the value of b).

In addition, b_{upper} increases faster than the b_{opt} as the ratio increases, given in Equation 5.9 and shown in Figure 5.7 as shown by the dotted lines. This means that as the ratio increases there is a wider range of b values which result in a lower execution time than recursive doubling ($b = 1$). This enables greater representation of larger composite numbers of processes and increases the size of the available space of schedules. Also, as b_{opt} remains significantly lower than b_{upper} the flexibility of the method increases even though the optimal value is small.

The b_{upper} value is determined as follows:

$$
\begin{aligned}
T_{\text{RM}}(b_{\text{upper}}) &= T_{\text{RM}}(1) \\
(\alpha_p + b\alpha_r)\log_{b+1} N &= (\alpha_p + \alpha_r)\log_2 N \\
(c+b)\log_{b+1} N &= (c+1)\log_2 N \;\mid\; c = \frac{\alpha_p}{\alpha_r} \\
\log_{b+1} 2 &= \frac{c+1}{c+b}
\end{aligned}
$$

Again, using the Lambert W function, we solve for b_{upper} resulting in Equation 5.9. As shown, b_{upper} is also independent of N. This allows us the flexibility to find factorisations of the process count purely dependent on the machine parameters. Equation 5.9 is plotted for a range of ratios of machine parameters in Figure 5.9.

$$
b_{\text{upper}} = -\frac{c+1}{\ln 2} W_{-1}\left(\frac{-\ln 2}{c+1} 2^{\frac{c-1}{c+1}}\right) - 1 \;\mid\; c = \frac{\alpha_p}{\alpha_r} \tag{5.9}
$$

Rounding b_{opt} for a given machine allows the usage for the *recursive multiplying* algorithm for AllReduce with a time cost given in Equation 5.10. This requires N to be power of $b+1$.

$$
T_{\text{RM}} = \log_{\lfloor b_{\text{opt}} \rceil + 1} N \left(\alpha_p + \lfloor b_{\text{opt}} \rceil \alpha_r\right) \tag{5.10}
$$

In reality N will rarely be a power of $b+1$ since the powers quickly become sparse on the number line. This is especially true when powers of $b \gg 1$ are used.

With the insight of *recursive multiplying* being a b-ary multicast building block, applied k-ary times, we can change the value of b in successive stages of the AllRe-

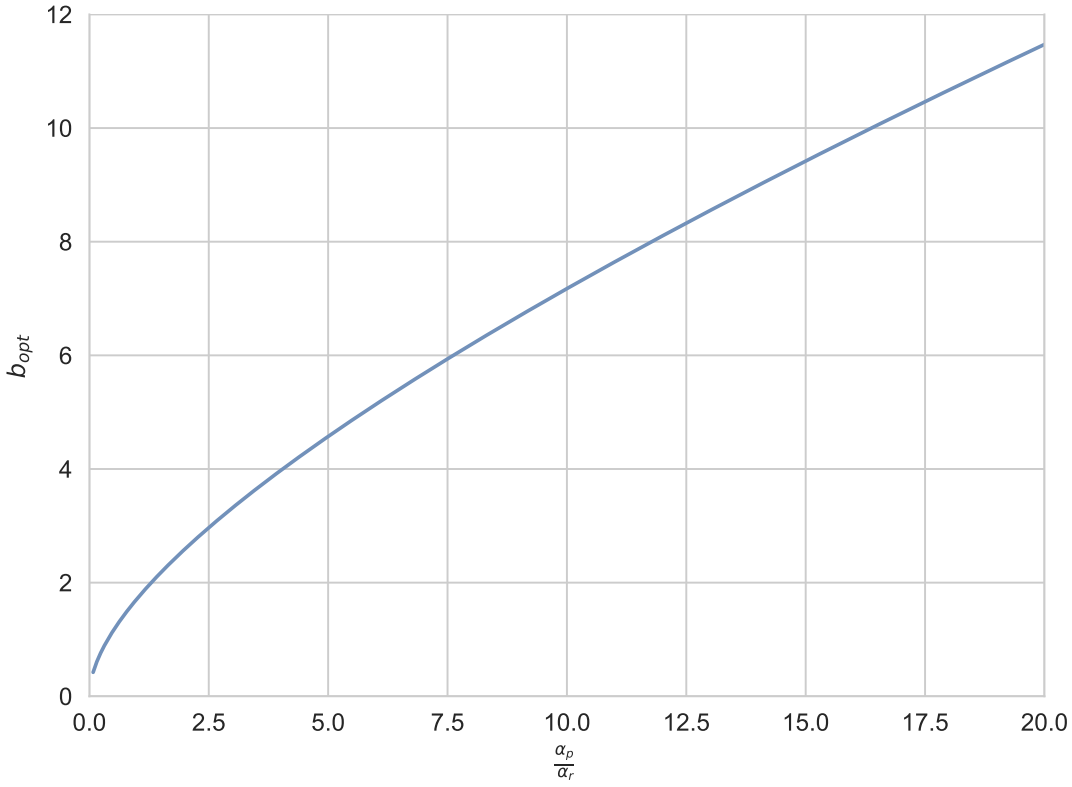


Figure 5.8: Illustration of the b_{opt} for a given ratio of machine parameters.

duce. This is illustrated in Figure 5.10. Being able to change the b parameter yields the flexibility of recursive multiplying.

Recursive doubling is restricted to the domain of powers of two, but in comparison, recursive multiplying has a much larger domain which contains the powers of all values of $b+1$ in addition to any product of different integer b values.

Through this mechanism we can find a schedule for any process count N by using the prime factorization of N. An optional improvement is to find a factorization which includes only numbers close to b_{opt}. Table 5.1 shows the capability of being able to factor the process count.

Several limitations are still present using the *recursive multiplying* method. First, the machine is required to be able to pipeline messages and must have a large enough injection bandwidth: without this facility the recursive doubling method is the best possible for small messages. Second, due to the use of a multicast, the receiving processes are required to provide b receive buffers, which cannot be overlapped since multiple receives occur per stage. The memory required is reflected in the specific factorization chosen, which could be used as a determining factor for the choice. Finally, if the

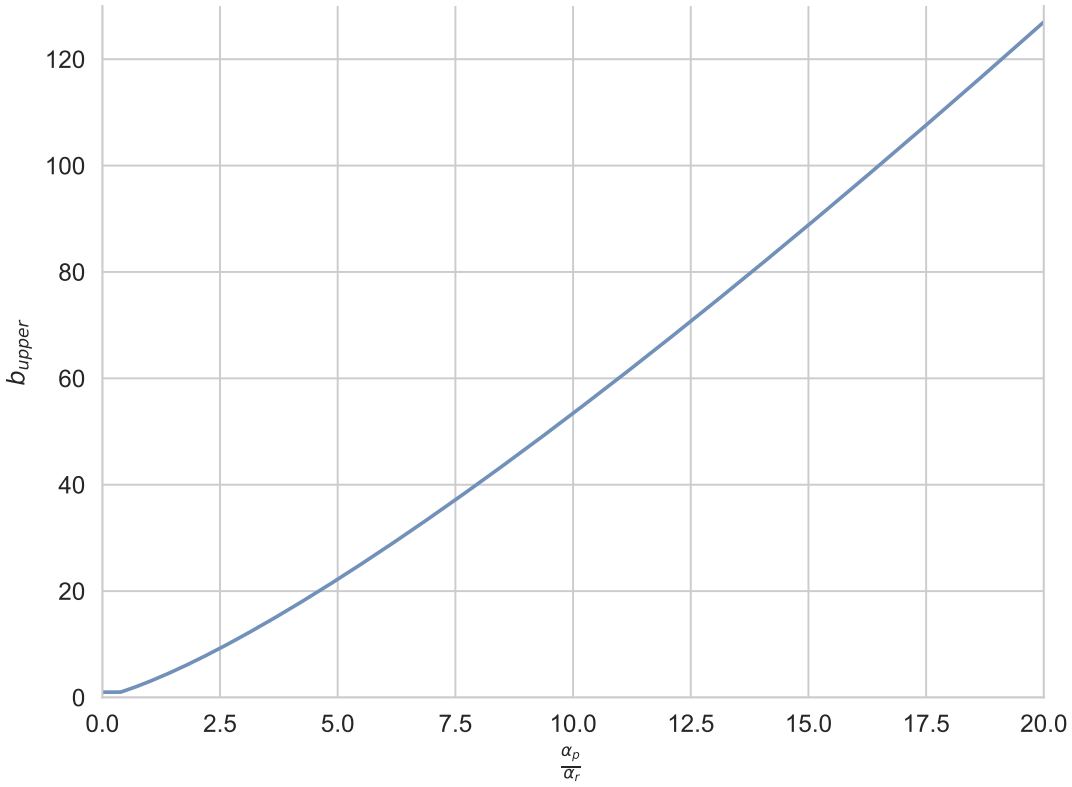


Figure 5.9: Illustration of the b_{upper} for a given ratio of machine parameters.

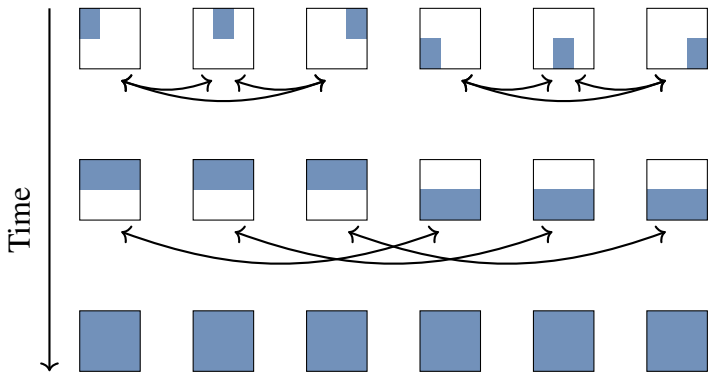


Figure 5.10: AllReduce operation with six processes using traditional illustration.

Process Count	Factorisations
10	(5, 2)
16	(2, 2, 2, 2) (4, 4)
36	(4, 3, 3)
48	(4, 4, 3)
60	(5, 4, 3)
80	(5, 4, 4)
1024	(2, 2, 2, 2, 2, 2, 2, 2, 2, 2) (4, 4, 4, 4, 4) (8, 8, 8, 2)

Table 5.1: Factorisation usable for the given process counts through the recursive multiplying method. Factorisations which contain only *2* are able to be used by recursive doubling. Not all factorisations are given for the respective process counts.

process count is not able to be factorized to suitably small numbers, a large multicast operation is required to take place, if no other additional method is used. This can inhibit the performance for *recursive multiplying* significantly.

5.4.2 Values Outside The Domain

While recursive doubling and *recursive multiplying* address certain process counts well, other process counts are not able to be used at all. In practice, the recursive doubling method is combined with a collapse and expand method to address non-powers of two process counts.

Figure 5.11 illustrates the methodology of the collapse and expand method to enable non-powers of two process counts to be handled. As an initial step, before recursive doubling is used, processes send their contributions to the AllReduce to a peer, such that the resulting active peers are of a power of two count. After recursive doubling is performed, the expansion phase takes place in which the previous peers send the result of the AllReduce to their respective peers, in order to satisfy the requirement that the final reduced vector is present on all processes.

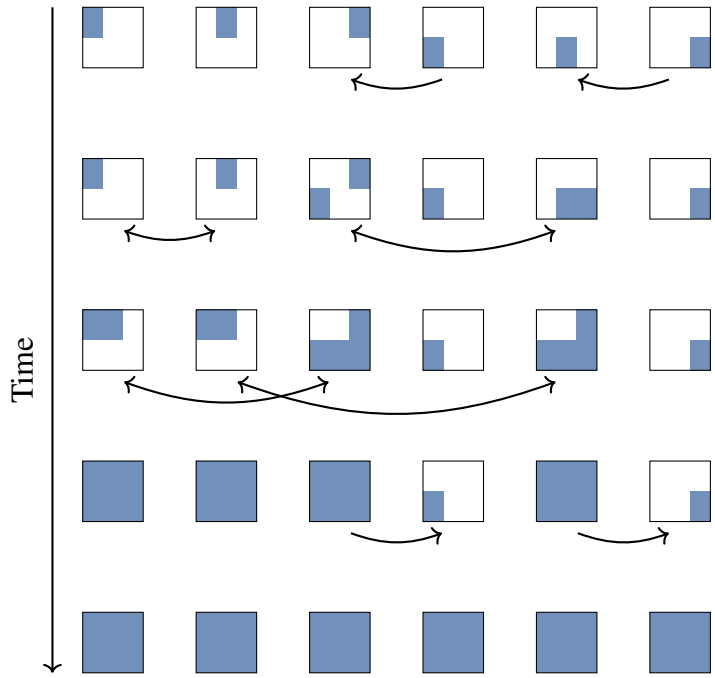


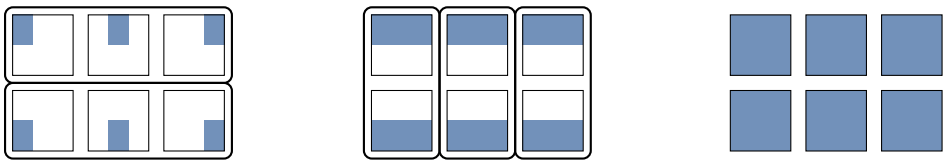
Figure 5.11: AllReduce operation with six processes using collapse and expand stages to allow a non-power of two processes AllReduce using recursive doubling.

Recursive multiplying has a much larger domain for process counts since the b parameter can be chosen at will. In combination with the capability of being able to switch the b value per stage, this enables a vast majority of process counts to be covered. However recursive multiplying fails to address prime numbers which are larger than b_{upper} and therefore would be split across multiple stages if it were possible.

Similar to recursive doubling, the collapse and expand method can be used to fix such a situation in which a large prime number is a factor of the process count. This has the same downsides, in that it adds two additional communication stages until the AllReduce operation is complete.

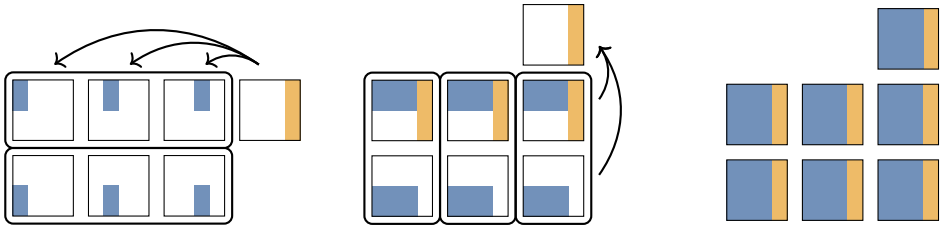
Using the pipelining latency-bandwidth model, with all implications of the analytical model, we calculate the collapsing stage to cost $\alpha_p + \alpha_r$, since all processes which become inactive send their buffers to the active processes. The expansion stage is the more expensive, because a single process needs to perform a multicast. The expansion stage is of $\alpha_p + (m-1)\alpha_r$ cost, where m is the collapsing factor. In Figure 5.11 the m factor is 2.

The fundamental idea of recursive multiplying is to exploit the capability of pipelining and therefore being able to send multiple messages. This idea can be used further to address the case of large primes. By viewing the factorization of the process count



(a) Stage 1 (b) Stage 2 (c) Result

Figure 5.12: Hypercuboid view of recursive multiplying for an AllReduce operation with six processes.



(a) Stage 1 (b) Stage 2 (c) Result

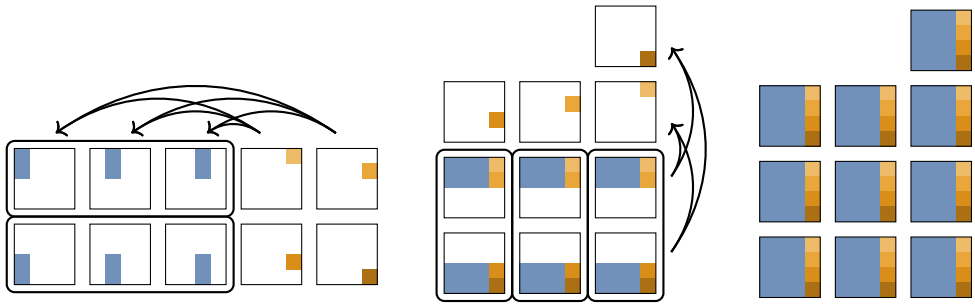
Figure 5.13: Hypercuboid view of recursive multiplying with prime merging of seven processes.

as a d-dimensional hyper-cuboid, as shown in Figure 5.12 where each stage is represented by a dimension, we can formulate the intuitive view of the AllReduce operation. The recursive doubling algorithm would be a binary hypercube in this illustration.

An alternative approach to handling large primes is to do *merging*, which does not require two additional stages. This allows a composite number to be used instead of a multiple of a base. The first stage is executed performing the first stage of the factorization, while the exposed remainder processes broadcast their own contribution to all processes as required. During the final stage all processes of a group send their final value to the remainder processes which then reduce these independently.

To ensure all processes obtain the same result, every element in the group must combine the same partial values in the same order. During the last stage, the remainder processes can also do this as they have the same set of partial results. However, unlike the processors in the main groups, they do not contribute to the set of partial results in that stage.

By decomposing the size of the AllReduce operation into two numbers, one of which is easily factorisable, we can make efficient use of multicast. An important optimization is to spread the remainder processes across the groups in the final stage, otherwise a single group will send many more messages and the method incurs a larger



(a) Stage 1 (b) Stage 2 (c) Result

Figure 5.14: Hypercuboid view of recursive multiplying with prime merging of ten processes. This merging pattern is purposefully chosen to illustrate the decomposition of remaining processes over the reduction groups to minimize the number of messages which are sent. The decomposition choice is crucial for performance of the prime merging method. For clarity some arrows are omitted.

overall cost. For prime merging to be used, the factorisation must contain at least two factors: this method cannot be used with a single stage.

In the given example shown in Figure 5.13, only two stages are required to perform an AllReduce across seven processes. The first stage consists of the first six processes performing an AllReduce within two groups, while the seventh process broadcasts its value to an entire group at the same time. This allows all members of that group to calculate the reduction with the contribution of the seventh process. During the second, and final, stage three groups of two processes perform a pairwise exchange, while a single group also sends its partial results to the seventh process to reduce them by itself. By using prime merging, we enabled the seventh process to receive all required information within the two stages, instead of the four required by the generalised fix.

Figure 5.14 illustrates a prime merging procedure with a larger number of remaining processes. The distribution of the remaining processes across the reducing groups is important for performance, otherwise a single group will be required to send many more messages which will induce a large skew.

$$T_{\text{MERGE\_INV}} = \begin{cases} \alpha_p + \alpha_r(b + r \operatorname{div} g) & r \bmod g = 0 \\ \alpha_p + \alpha_r(b + r \operatorname{div} g + 1) & \text{otherwise} \end{cases} \tag{5.11}$$

Viewing the prime merging method with the pipelining latency-bandwidth model, we can evaluate the cost of the merge phase as $\alpha_p + \alpha_r(b+1)$. The multicast cost from the remainder processes is always larger than the internal AllReduce multicast.

For the final stage of the prime merging method the cost is given in Equation 5.11 as $T_{\text{MERGE\_INV}}$. The remainder process count is r and the group size is g. The cost is a summation of the AllReduce performed internally in the core group, and the multicast to the remainder processes. The piece-wise function is used to denote the division of remainder processes across the existing internal groups. The additional single α_r cost is due to the maximum between the groups with additional remainder processes than others, if the remainder processes do not evenly divide.

5.4.3 Implementation

The pseudocode for recursive multiplying is presented in Algorithm 2. The pseudocode shows several required generalisations, compared to the recursive doubling, presented in Section 5.2.2.2. The transformation from stage ranks to global ranks is done similarly with a branching statement, though the transformation is more complex than the power-of-two case. In addition, the masking to find the relevant peers for a stage has changed from an exclusive-or operation to a group and offset calculation. Finally, the mask incrementing has changed from a left shift operation, to a multiply by the stage base.

For our prototype, we chose to pass the schedule directly to the AllReduce operation globally and not compute the schedule on the fly. This would enable library implementors to have control over which schedules are used when. The schedules consist of instructions which some stages will execute depending on previous stages.

Schedules are a tuple of instructions which execute the correct data movement in sequence in order to result in an AllReduce operation. An implied reduction phase is present with each instruction in the schedule. The simplest schedule component is the all-to-all instruction of form aB. The B parameter is the reduction factor for all groups. The processes are grouped such that $B = \frac{N}{groups}$. The schedules as implemented are an example of possible ways in which to enable the use of recursive multiplying in a flexible fashion, but MPI library implementors can implement recursive multiplying in other ways.

The collapse and expand method is implemented using two additional stages at the beginning and end of the factored schedule. Two variables are used to determine the behaviour of the collapse. The first is the *threshold*: this determines which processes are part of the collapse method, either as receivers or senders. If the rank is below the threshold, the process determines whether it is a sender or receiver.

Algorithm 2 Recursive Multiplying AllReduce

```
1:  procedure ALLREDUCE(rank, com, schedule)
2:      value ← com                                          ▷ initialize variables
3:      stage_mask ← 1
4:      pthres ← 0
5:      pbase ← 1
6:      wid ← rank

7:      for stage in schedule do
8:          if type(stage) is factor then
9:              sfactor ← factor                             ▷ recursive multiplying
10:             sbase ← sfactor × stage_mask
11:             if wid ≠ −1 then
12:                 for index ∈ [0, sfactor−1) do            ▷ find peers
13:                     mask ← (index +1) × stage_mask
14:                     block ← ⌊wid/sbase⌋ × sbase
15:                     offset ← (wid + mask) mod sbase
16:                     peer ← block + offset
                                                             ▷ stage rank to global rank
17:                     if rpeer < pthres then
18:                         rpeer ← peer × pbase + pbase −1
19:                     else
20:                         rpeer ← peer + (pthres/pbase) × (pbase −1)
21:                     Send non-blocking value to rpeer
                                                             ▷ complete stage
22:                 for peer ∈ [0, sfactor) do
23:                     Recv value from peer
24:                     Reduce value rbuf
25:                 Wait on sends
26:             stage_mask ← stage_mask × sfactor
```

This is done using the *base* variable. The *threshold* must be divisible by the *base*. The *base* is effectively the collapse ratio, which determines how the processes beneath the threshold are combined. The expansion stage, the final stage, is the inverse operation of the collapse stage. The processes which are collapsed into other processes are

Algorithm 3 Recursive Multiplying Collapse

```
27:  else if type(stage) is collapse then
28:      pthres, pbase ← collapse
29:      if rank < pthres then
30:          if rank (mod pbase) ≠ (base−1) then
31:              peer ← ⌊rank/pbase⌋ × pbase + pbase − 1
32:              Send value to peer
33:              wid ← −1
34:          else
35:              Recv rbuf from peer
36:              Reduce value rbuf
37:              wid ← ⌊rank/pbase⌋
38:      else
39:          wid ← rank − pthres/pbase × base − 1
```

Algorithm 4 Recursive Multiplying Expansion

```
40:  else if type(stage) is expand then
41:      pthres, pbase ← expand
42:      if rank < pthres then
43:          if rank (mod pbase) = (base−1) then
44:              for b do
45:                  peer ← wid × pbase + b
46:                  Send non-blocking value to peer
47:          else
48:              Recv value from peer
49:          Wait on sends
```

Algorithm 5 Recursive Multiplying Merge

```
50:     else if type(stage) is merge then
51:         remainder, groups, factor ← merge
52:         group_size ← factor × stage_mask
53:         if rank < remainder then
54:             wid ← −1 − rank
55:             group ← rank (mod groups)
56:             group_first ← remainder + group × group_size
57:             for each process in group do
58:                 peer ← group_first + idx
59:                 Send non-blocking value to peer
60:         else
61:             wid ← rank − remainder
62:             group ← ⌊wid / group_size⌋ × group_size
63:             for each peer in group do
64:                 Send non-blocking value to peer
65:             Wait for all receives
66:             Reduce all buffers
67:         Wait on sends
68:         stage_mask ← stage_mask × factor
```

Algorithm 6 Recursive Multiplying Inverse Merge

```
69:     else if type(stage) is invmerge then
70:         remainder, groups, factor ← invmerge
71:         groupsize ← factor × stage_mask
72:         if rank < remainder then
73:             Wait for all receives
74:             Reduce received buffers
75:         else
76:             group ← ⌊wid / group_size⌋ × group_size
77:             for each peer in group do
78:                 Send non-blocking value to peer
79:             for each remainder do
80:                 if remainder (mod groups) = wid (mod groups) then
81:                     Send non-blocking value to remainder
82:             Reduce all buffers
83:             Wait for sends
84:     Return value
```

inactive during the rest of the collective operation, until the expansion stage. The pseudocode for the collapse and expand stages is shown in Algorithm 3 and Algorithm 4.

Similar to the collapse and expand method, the merging method is applied at the beginning and end of a schedule, but it enables additional overlapping of communication and computation with the remaining processes which are not within the core processes. The merging method is characterized by three parameters which uniquely define how the method is applied.

The first variable is the number of processes to be merged into the internal processes, called the *remainder*. This number defines how many processes send their contribution to the AllReduce to all processes within their group, according to their rank. The other two parameters are the *groups* and the *factor*. These two variable define the face of the hypercuboid into which the *remainder* processes are merged. The pseudocode for the merging method is shown in Algorithm 5 and Algorithm 6.

Since $remainder + groups \times factor = N$ is always true, the *groups* variable is not explicitly required and can be calculated on the fly. We chose to label this explicitly, because it simplifies the inverse merge operation at the end of the stages.

Using the above described schedule components, it is possible to construct a large number of schedules which will be addressed in Section 5.4.4. We use the collapse and expand, and merging, stages explicitly at the beginning and end of a factored schedule which reduces the possible space of schedules. However both the collapse and expand, and the merging, stages could be used during a schedule as well.

This may be beneficial, because the algorithm would be performed across a smaller number of cooperating processes in their respective groups, thereby avoiding congestion, which is rarely modelled, but often encountered in real-world applications. Allowing for collapse and expand stages within a schedule will increase the number of schedules for a given process count vastly.

5.4.4 Heuristic Schedules

Determining the schedule with which a specific AllReduce operation should be executed is non-trivial. Due to the vast choice of schedules for anything but the smallest of process counts, it is difficult to choose a schedule quickly which yields the lowest latency. Figure 5.15 illustrates the number of possible schedules for the first 1024 process counts. The majority of the schedules available for a specific process count are not efficient, as their execution time will be longer than recursive doubling. In addition,

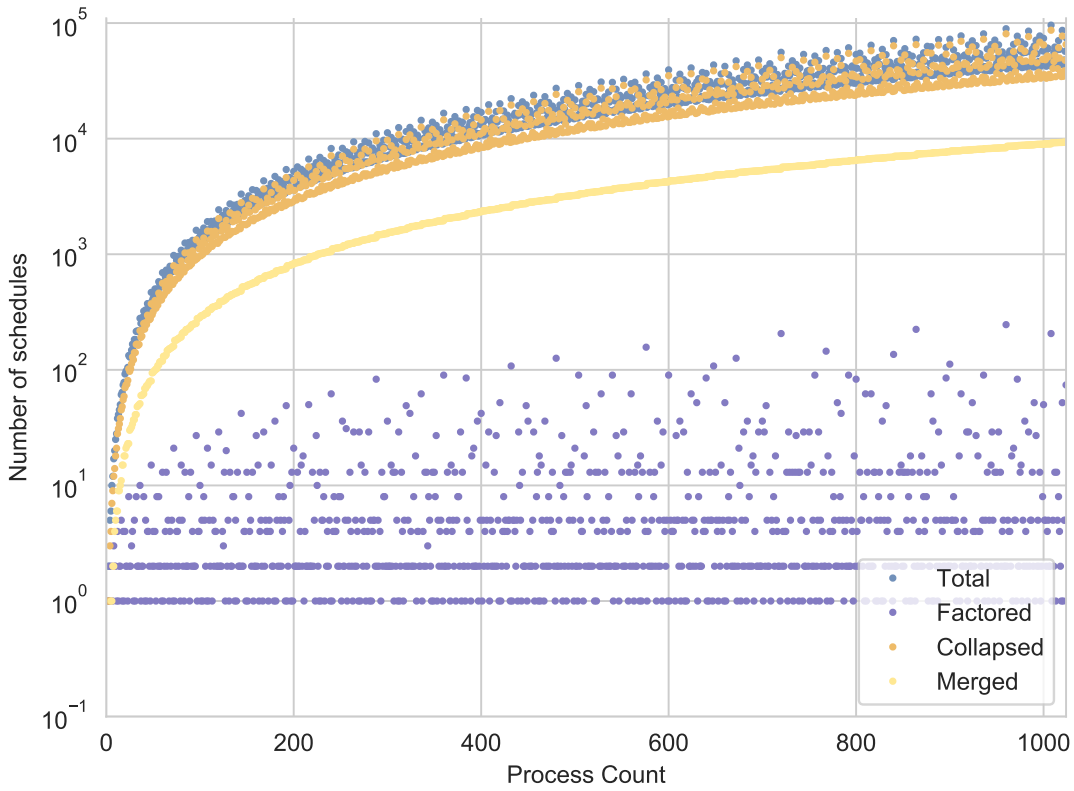


Figure 5.15: Illustration of the number of schedules of three categories available for the first 1024 process counts. The *Factored* category is for schedules which are a factorization of a process count. The *Collapsed* category is for schedules which utilize the collapse/expand method and the *Merged* category is for schedules which utilize the prime merging method. Permutations of factored schedules within merged schedules are ignored.

the choice of schedule needs to be performed quickly and effectively.

One way to handle the selection would be to measure all schedules for all possible process counts and store the best schedule as a lookup table for the implementation to fetch when required. This has the downside that, for large machines, the amount of memory required is large. In addition, this memory is required to be accessed by all participating processes, which can be detrimental to performance.

We have developed a heuristic function, shown in Algorithm 7, which determines a good schedule to be used for a given process count. The heuristic is based on the idea that the b_{opt} value is the factor with which we want to factor the process count, but numbers near b_{opt} are acceptable. Specifically numbers up to b_{upper} are considered, given the flatness of the functions shown in Figure 5.7, since these perform equally

Algorithm 7 Heuristic Schedule Generation

function HEURISTIC($\alpha_p, \alpha_r, count, b_{\text{upper}}, merge \leftarrow 0$)
  $product \leftarrow 1$
  $divisors \leftarrow empty\ list$

  $potential \leftarrow$ SORT($[2, b_{\text{upper}} + 1]$ ascending w.r.t. $(\alpha_p + b\alpha_r) \log_{b+1} count$)

  for *divisor* in *potential* **do**
    while $count \bmod (product \times divisor) == 0$ **do**
      $divisors \leftarrow divisor$
      $product \leftarrow product \times divisor$

  if $count \neq product$ **then return** HEURISTIC($\alpha_p, \alpha_r, count - 1, b_{\text{upper}}, merge + 1$)
return (*divisors*, *merge*)

well or better in the model compared to $b = 1$.

The heuristic is initiated by sorting the range of numbers between 2 and $\lfloor b_{\text{upper}} \rfloor + 1$ according to Equation 5.7 in non-integer space. This prioritizes values of b which are predicted to have a lower overall time. The heuristic attempts to factor the process count by the *divisor*, successively reducing the factor to capture as many factors close to b_{opt} as possible. Effectively, the heuristic explores the numbers which may be factors along the dotted lines shown in Figure 5.7 in a prioritized manner.

An alternative to using a sorting algorithm is to iterate the range from $\lfloor b_{\text{upper}} \rfloor + 1$ to 2, inclusive, and use each value as a divisor. This causes schedules to be chosen which contain factors which are larger than b_{opt} and therefore is less effective, but avoids the overhead of a sorting implementation.

If the final factor is too large, determined by the $\lfloor b_{\text{upper}} \rfloor + 1$ parameter, then the merging method discussed in Section 5.4.2 is used to reduce the process count and the heuristic is applied to the $N - 1$ value. A satisfactory factoring and merging combination is found when all factors are below or equal to the $\lfloor b_{\text{upper}} \rfloor + 1$ limit.

Figure 5.16 shows the efficiency of heuristically chosen schedules compared to the best possible within the model explored in Chapter 4. The heuristic often chooses the best schedule as shown by matching the *best schedule* line. The recursive doubling schedules are shown as a point of comparison.

For the first 1024 process counts, the recursive doubling method is outperformed

Process Count	Heuristic Schedule	Efficiency (%)	Best Schedule
11	(11)	91.6	(3, 3) + 2
19	(6, 3) + 1	93.3	(4, 4) + 3
22	(11, 2)	88.1	(5, 4) + 2
23	(11, 2) + 1	78.7	(5, 4) + 3
29	(4, 7) + 1	94.1	(5, 5) + 4
33	(3, 11)	94.4	(5, 6) + 3
34	(3, 11) + 1	84.9	(6, 5) + 4
41	(4, 5, 2) + 1	95.1	(6, 6) + 5
43	(6, 7) + 1	99.5	(5, 4, 2) + 3
44	(4, 11)	95.5	(5, 4, 2) + 4

Table 5.2: Generated schedules using the heuristic under $N = 50$ for which the efficiency is below 100%.

by the heuristically chosen schedule. The heuristic uses the $upper = 12$ given by Equation 5.9 and the measured machine parameters from Chapter 4. We use the average of all the efficiencies for the first 1024 process counts for a given method as a measure of effectiveness for the method. The recursive doubling method achieves an average of 72.1% efficiency, while the non-sorted heuristic achieves 92.4%. This is improved upon by the sorted heuristic approach to 97.1%.

Table 5.2 shows the process counts below $N = 50$ for which the heuristically generated schedules are not the best schedules. These are also represented in Figure 5.16 as the first fifty heuristic schedules. The schedules show the slight deficiency of the heuristic, which is that schedules which are able to be factored from the process count are always chosen before a merged schedule. This is due to the greedy, breadth first, approach in exploring the factorization tree.

5.4.5 Large Messages

Similar to Section 4.3.2, this section will discuss larger message AllReduce operations, but will not fully explore the subject. The Equations 5.7, 5.8, 5.9, and 5.10 do not apply directly to operations with larger message sizes, since the bandwidth and computational terms are ignored from the original latency-bandwidth model, but they are used as an approximation, since the bandwidth and computational terms can be neglected for less than medium sized messages. For ARCHER, taking into account the

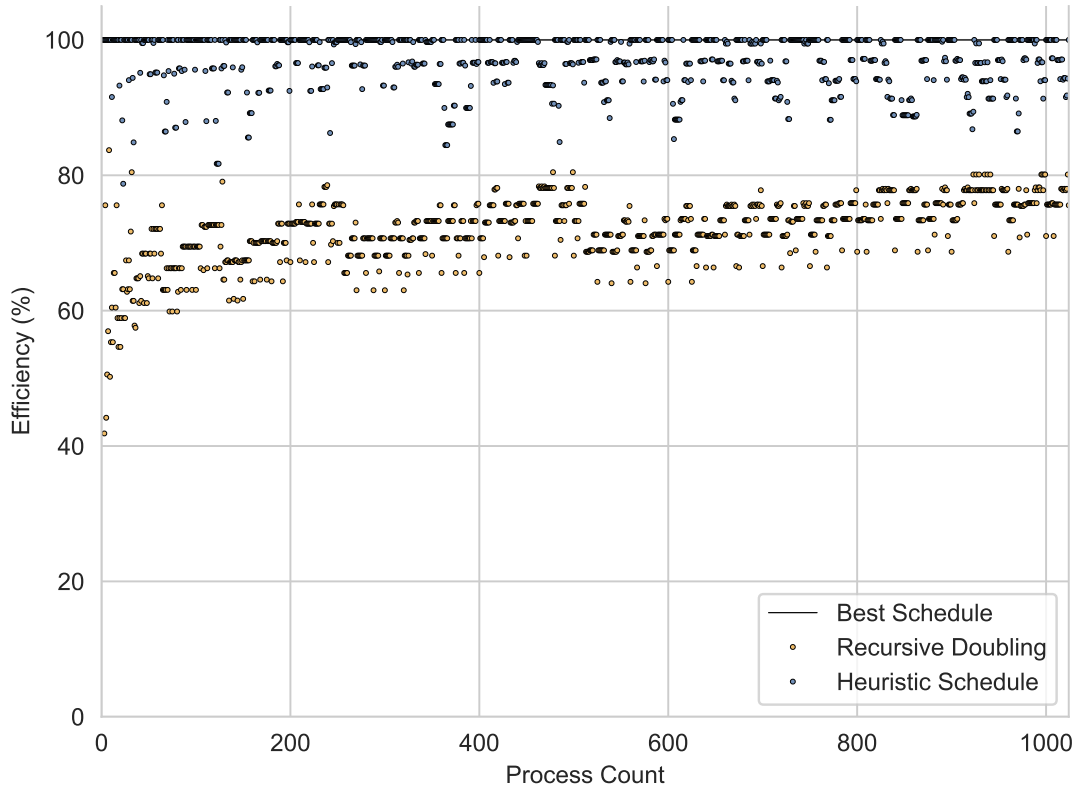


Figure 5.16: Efficiency plot of schedules generated using the heuristic compared to the lowest latency schedules found through exhaustive searching using the theoretical model. The recursive doubling schedules are shown as a point of comparison. The machine parameters measured in Chapter 4 were used to calculate the timings for all schedules.

results in Table 4.1, we can consider medium sized messages to be above 512 bytes.

The finding from Figure 4.8 is the ratio $\frac{\alpha_p}{\alpha_r}$ tends to zero as the message size increases, as also shown in Table 4.1. If we consider Figure 5.8 and Figure 5.9, we can see that a decreasing ratio causes the range of available b values to be reduced.

The implication of this reduction is not that the larger message sized AllReduce operations cannot be performed with recursive multiplying, but that there are fewer available schedules which reduce the execution time compared to recursive doubling. When the machine parameter ratio is $\frac{\alpha_p}{\alpha_r} = 0$, the only applicable value is $b = 1$, which resolves to the recursive doubling method. In other words, recursive doubling is the most bandwidth optimised schedule possible with *recursive multiplying*.

5.5 Experimental Results

5.5.1 Environment

All the experiments reported in this Section were run on the ARCHER[32] supercomputer, a Cray XC30 machine with 4920 compute nodes, each with two 12-core Intel E5-2697 v2 CPUs. The interconnect is the Cray Aries in a Dragonfly topology. The environment used was:

- PrgEnv-cray/5.2.56
- dmapp/7.0.1-1.0502.10246.8.47.ari
- cray-mpich/7.2.6
- pmi/5.0.7-1.0000.10678.155.25.ari
- ugni/6.0-1.0502.10245.9.9.ari

In all cases we used one rank per node, so that all communication is over the network and not in shared memory on the node. All measurements are performed using an AllReduce summation operation of a single 8 byte integer.

Applications commonly use a single MPI process per core on a node, *flat mode*, which on modern machines would result in tens to hundreds of processes communicating internal to a node. MPI implementations typically use shared memory transport mechanisms without traversing the network at all for these exchanges. In addition the MPI libraries would optimize an on-node and off-node algorithm choice for an AllReduce operation: we address only the off-node component.

5.5.2 AllReduce Benchmark

The benchmark to evaluate the algorithm presented in Section 5.4 is implemented using the Cray DMAPP library[22], which supports a PGAS-based approach to communication. Although the algorithm presented does not explicitly require single-sided communication, using Cray DMAPP allows the least amount of time between message issues without a large software stack, which enables us to maximize the message pipelining.

The memory consumption is less efficient than a point-to-point channel implementation. Both approaches are difficult to quantify: point-to-point channels consume $O(1)$ memory, but $O(\log_2 N)$ channels exist in memory. The PGAS-based implementation utilises a memory array allocated in the data segment of the application, which stores the addresses to write to for each peer.

The live ARCHER system was used for measurements, therefore noise is present throughout all results. The experimental setup is to measure all results in blocks of 10 AllReduce operations. The block-size is further explored in Section 5.5.6. This is done to limit the effects of the resolution of the timing routines and to reduce, or average, skew effects present in the measurements. The number of blocks is the same for each node allocation and set at 250. This is to ensure a large number of samples on different node allocations, but work within budget constraints.

The node allocations given on the live ARCHER system are variable and dependent on job requirements. Therefore at least forty node allocations were taken, then an evaluation of the results was performed and more node allocations were measured if the change in median and mean was not below 5% compared to the prior median and mean, respectively. This allows measurement of all potential noise sources such as hardware failures, OS noise, network noise and system load.

5.5.3 AllReduce Schedule Comparison

We compare the performance between an implementation of recursive doubling and an appropriate schedule for a given collective size. The schedules used to represent recursive doubling were exactly the behaviour which would be performed in MPICH. The power-of-two cases were handled by a series of *a2* stages. The non-power-of-two stages were handled by collapse and expand stages with a series of *a2* stages between them.

The results for recursive multiplying used the schedules presented in Table 5.3. Re-

Process Count	Schedule	Blocks	Min RD/RM (μs)	Min %	Median RD/RM (μs)	Median %
4	a4	10500	2.36 / 1.87	21.1	4.55 / 3.65	19.7
6	a6	10250	4.76 / 2.87	40.0	8.72 / 5.75	34.1
8	a2,a4	10750	4.37 / 3.54	18.9	9.38 / 7.23	23.0
12	a3,a4	10000	7.03 / 4.43	37.0	15.3 / 11.6	24.4
16	a4,a4	12500	6.85 / 4.98	27.3	19.1 / 13.8	27.6
24	a4,a6	13750	9.99 / 6.91	30.8	29.7 / 25.5	14.2
32	a8,a4	10000	12.9 / 8.95	30.8	30.9 / 25.2	18.4
48	a8,a6	10000	19.8 / 13.3	33.2	38.7 / 32.1	17.1
64	a8,a8	10000	24.2 / 16.5	31.9	47.4 / 39.9	15.9
96	a8,a3,a4	12500	25.3 / 20.7	18.1	51.7 / 47.6	7.96
128	a8,a4,a4	10000	25.1 / 17.9	28.9	101.0 / 88.4	12.5

Table 5.3: Percentage decrease in minimum and median execution times from recursive doubling to recursive multiplying.

sults for both the improvement on the minimum and median are shown, with decreases in execution time ranging from 8% to maximum improvements of 40%. The number of blocks measured for each process count is shown. All process counts were measured with at least 100000 AllReduce operations, due to the large variance on ARCHER. The jobs were allocated according to machine availability, this includes jobs which were spread across multiple chassis and groups (which included optical links).

The schedule choice was done experimentally by exhaustively measuring all schedules possible for a given size and then selecting the schedule with the minimal value of the median execution. This method of choosing which schedule to use is not representative of what would be done for each AllReduce execution. In production, the machine administrator would execute a benchmark which would evaluate all schedules and then statically assign this for a certain size, or utilise a heuristic as outlined previously.

The process count range was chosen to be within a reasonable experimental budget due to the large number of samples required on a machine such as ARCHER to find reliable results. This is not a limitation of the experimental results since in future it is expected that each node will contain more compute capability either with multiple GPUs or larger CPUs.

The performance results of the benchmark are presented in Figure 5.17. As can be seen, the minimum values are significantly less than the median values, with considerable spread of all measurements. Neither the minimum or median results follow

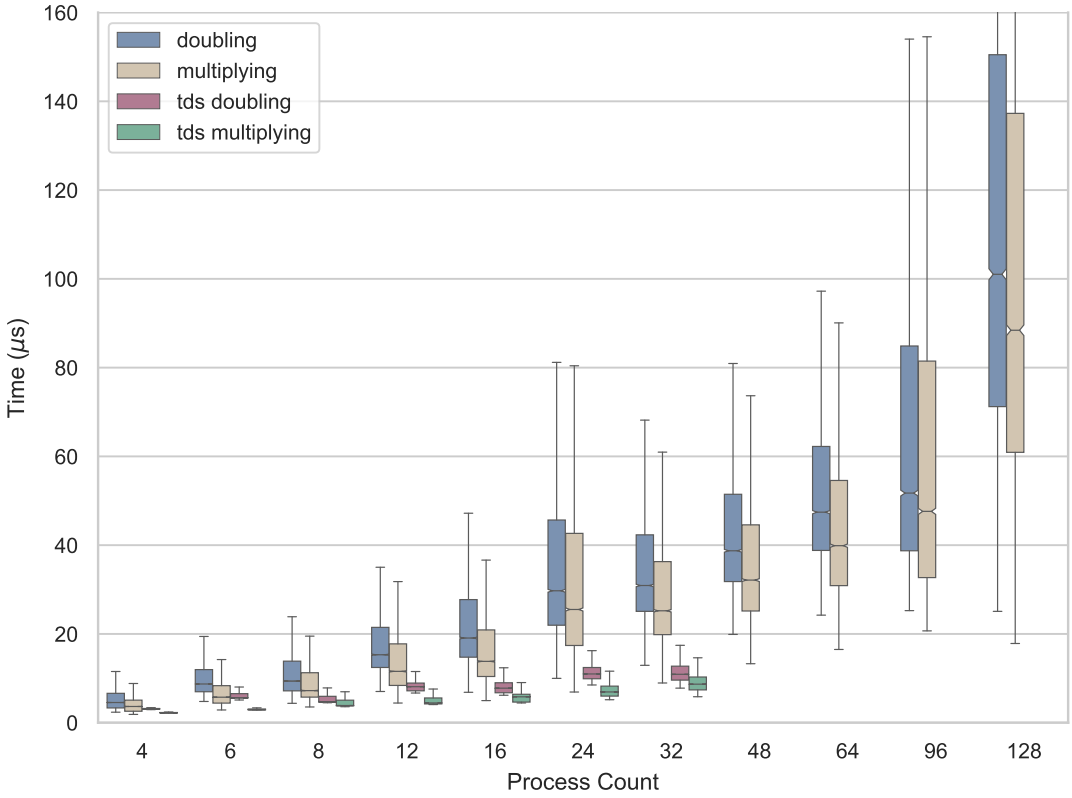


Figure 5.17: Execution times for varying sizes of AllReduce using both recursive doubling and recursive multiplying. The doubling and multiplying datasets were collected from ARCHER. The *tds* doubling and multiplying datasets were collected from the *tds* platform.

a logarithmic curve when going to large scales. Comparing recursive doubling to the best *recursive multiplying* schedule, there is a significant advantage by using message pipelining: the median value for recursive multiplying is near the 26th percentile value for recursive doubling. Important to note is the reduction in improvement as the scale grows, except for $N = 128$, for which the gains improve.

The *tds* results shown in Figure 5.17 are executions of the respective schedules of on a separate Cray XC30 machine used for testing and development of the ARCHER supercomputer. We used this cluster to understand what impact the congestion present on the large computer has on the execution of the algorithm, because we were able to have exclusive access to this machine. From the results, it can be seen that congestion is the major contribution to the execution times.

From Figure 5.17 we can see the schedules executed on both machines have significantly different results. The minimum and the spread of the time-to-solution for both the *recursive multiplying* and recursive doubling schedules are lower on the *tds* platform. With the only change between the two experimental job runs being the machine on which it was executed the conclusion is that the network congestion present on ARCHER is the source of this difference. Any self-congestion effects would still be present in the *tds* experimental results.

The *recursive multiplying* schedules intrinsically generate more messages per stage of the schedule compared to the recursive doubling schedule for a given process count. Using Equation 5.12 we can calculate the number of messages sent with any schedule, ignoring the collapse/expand and merging method. In the special case when a schedule consists of the same b values, Equation 5.13 can be used to calculate the number of messages.

$$\text{messages}(\text{schedule}, N) = \sum_{f \in \text{schedule}} N(f-1) \tag{5.12}$$

$$\text{messages}(\text{f}, N) = N(f-1)\log_f N \tag{5.13}$$

For example, with the process count $N = 64$ using the *recursive multiplying* schedule *(4, 4, 4)* would result in 576 messages being exchanged. With recursive doubling using the schedule *(2, 2, 2, 2, 2, 2)* the number of messages is 384. The recursive doubling schedule produces the least messages to exchange for all process counts, but has higher latency as shown in Figure 5.17.

Due to the increased number of messages to be sent by the *recursive multiplying*

method which the network needs to handle, congestion increases for the entire system. This would influence other jobs running nearby the job which is using the *recursive multiplying* method by decreasing the overall available bandwidth. More modern systems which have higher bisection bandwidth and better routing algorithms would be able to handle this increased demand in bandwidth better than ARCHER.

5.5.4 Message Size Scalability

The recursive multiplying algorithm was designed to improve the latency of small sized messages. To test how capable the algorithm is of accepting larger messages, we performed a message size sweep on both 8 and 64 process count executions. A subset of all possible schedules was chosen to be evaluated, which included the recursive doubling schedule. The schedules used are shown in the legends of Figure 5.18 and Figure 5.19 respectively. The message count is the count given if an MPI function call were executed. All messages are multiples of 8 bytes, so, for example, message count 8 corresponds to 64 bytes.

Figure 5.18 shows an expected behaviour, with a latency bound region and then a switch into a bandwidth bound region at a message count of approximately 24 to 32. The (a2,a4) schedule performs surprisingly well, with the minimum execution time being the best one for the entire sweep and the median being approximately equivalent to recursive doubling at higher message counts.

Figure 5.19 shows the message size sweep results for $N = 64$ executions. Due to running this benchmark later compared to previous results, the environment of the ARCHER supercomputer (both software versions and usage) has changed enough to see a strong difference in execution times. However, we are comparing only the data shown on the plot. As seen at the low end of the message count axis, recursive doubling is likely the best option, but as the message count increases the performance of recursive doubling is significantly worse than at the low end. At 512 message count (4096B) the recursive multiplying schedules clearly outperform the recursive doubling schedule. We cannot explain this result, since recursive multiplying was designed to perform well with small messages. We suspect that the adaptive network allows more bandwidth to be used, since the algorithm is sending many more messages for each stage and therefore puts more network load on surrounding paths.

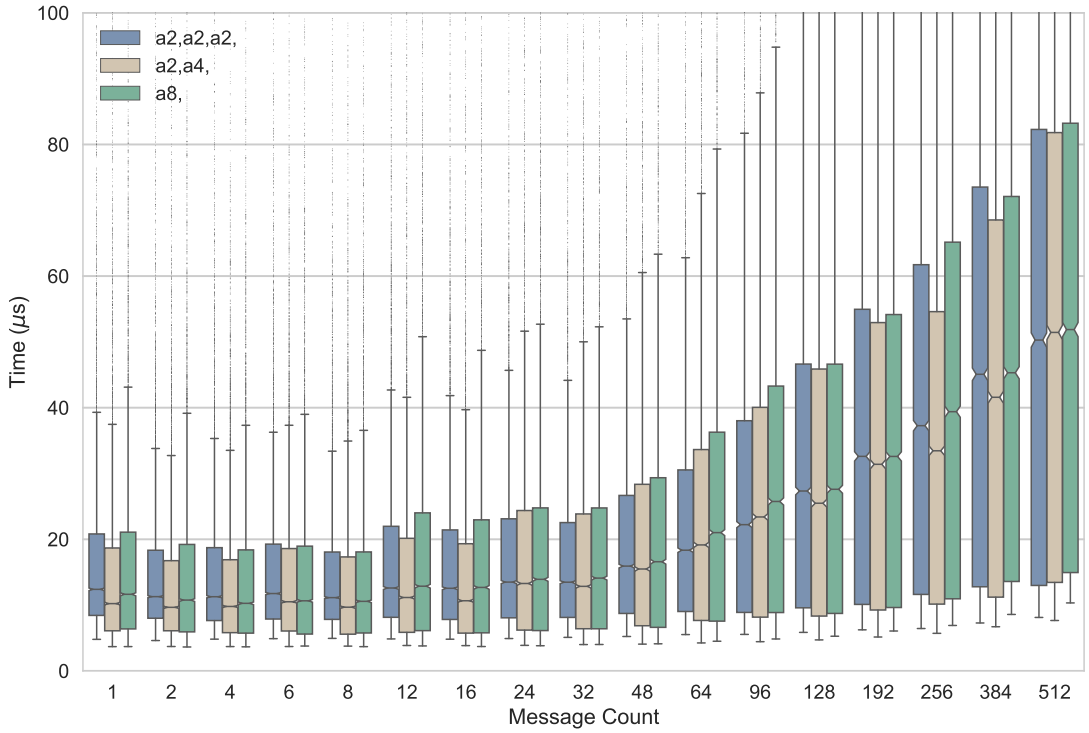


Figure 5.18: Executions times for varying message sizes using eight processes with all possible schedules for $N = 8$.

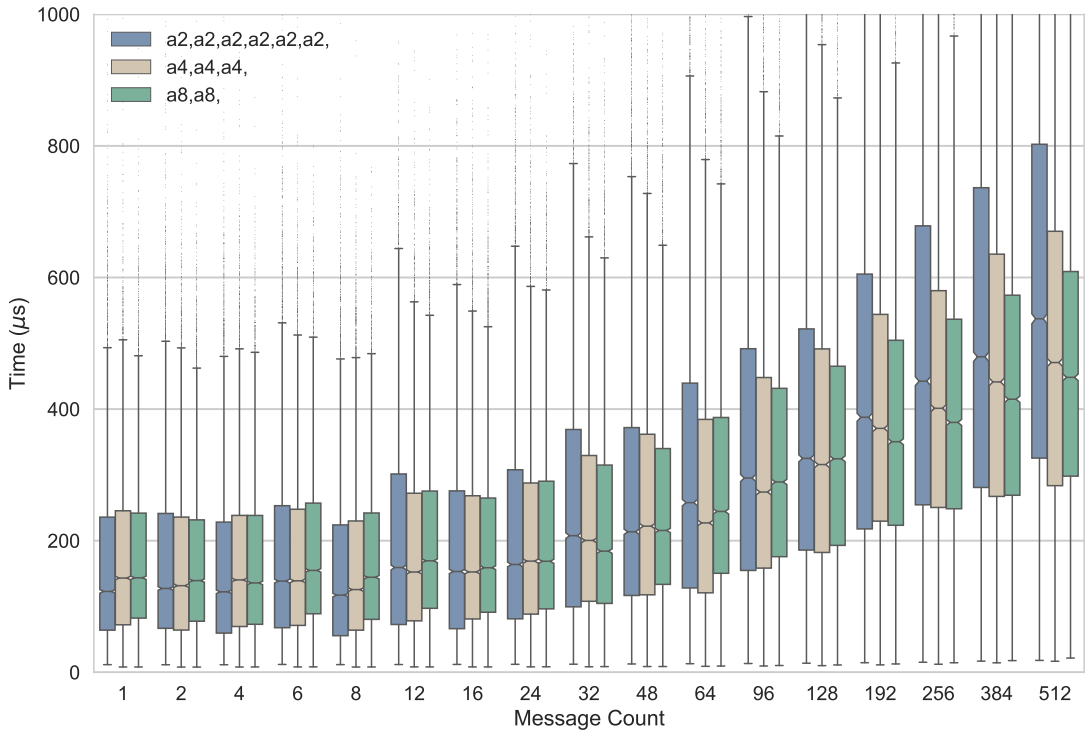


Figure 5.19: Executions times for varying message sizes using 64 processes with a subset of schedules chosen.

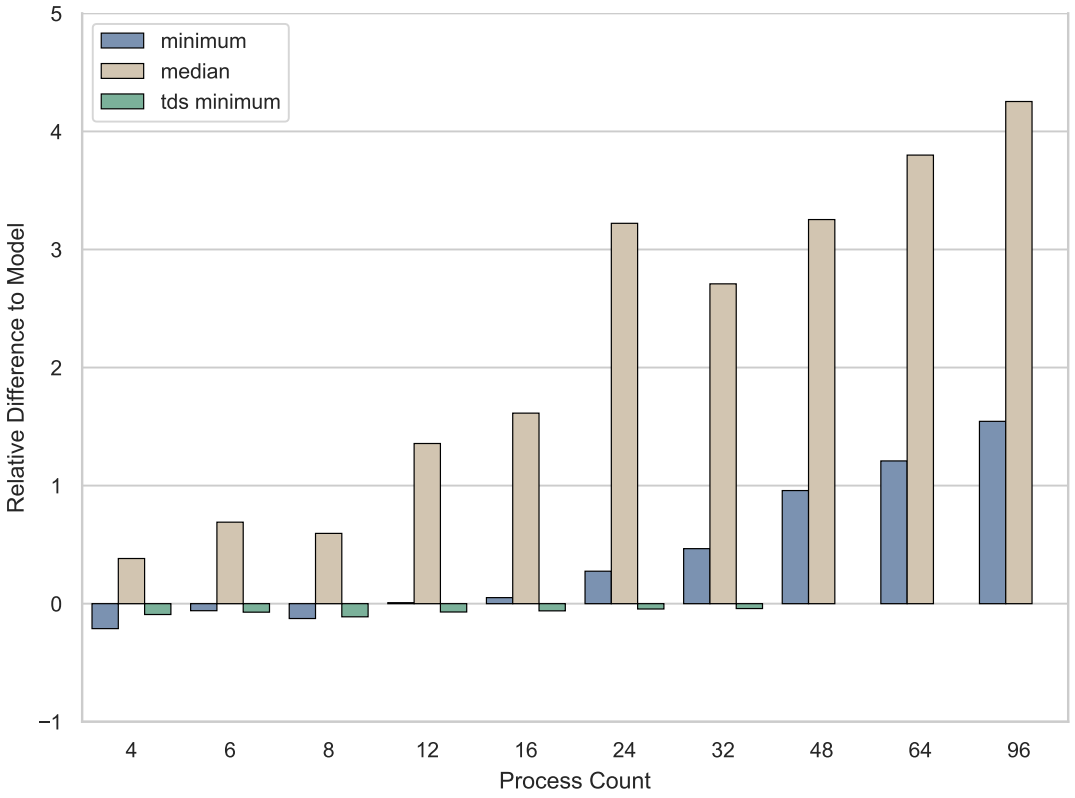


Figure 5.20: Relative difference of experimental minimum and median to prediction from pipelining latency-bandwidth model. The y-axis value is calculated using Equation 5.14.

5.5.5 AllReduce Model Comparison

The pipelining latency-bandwidth model presented in Chapter 4 can be used to predict the time required for an AllReduce operation. To measure the accuracy of this approach, we used the prediction of the model compared to the experimental results given in Section 5.5.3. Using the values for α_p and α_r evaluated previously for the median and minimum experimental distributions, we can use the model to predict the minimum and median values for AllReduce operations. We use the median as being more representative of the observed results to understand the pipelining latency-bandwidth model's predictive ability.

Figure 5.20 shows the relative difference, calculated with Equation 5.14, of the experimental results to the predictions by the model. As can be seen, the minimum values follow the model well until 24 nodes is reached. An upwards trend is clear from there onward for both the minimum and median values. The median values show a

consistent increase in the difference across all process counts.

$$\text{relative\_difference} = \frac{\text{experiment} - \text{model}}{\text{model}} \tag{5.14}$$

The model evaluation was performed using Equation 5.7 using the schedule which gave the shortest execution times in the experimental trials in Section 5.5.3. The machine parameters for the minimum model are taken from Section 4.3, while the median model parameters were found using a similar regression fit as Figure 4.7, but the median values instead of minimum values.

The difference between the theoretical and experimental values is likely due to skew of process arrival times. The process arrival time is the *global* entry time of individual processes to a single collective operation. In a theoretical context this is typically considered globally uniform, but in practice process start up times, system noise, network noise and various other factors can affect the skewing of progress within separate processes and therefore they arrive at the entry point at different global times. The skew increases as the size of the AllReduce increases, since it is more likely that any single process is delayed.

To improve the error observed in Figure 5.20 a more accurate model is required. From the relative errors shown the pipelining latency-bandwidth model clearly does not capture all underlying effects present in reality. The minimum values are modelled well up to approximately 16 nodes, but also diverge afterwards. One aspect not modelled, congestion, could be excluded in a *tds* environment, in which only a single job is run at any given time. However, modelling congestion would add significant complexity to the pipelining latency-bandwidth model.

To support the assertion that the pipelining latency-bandwidth model accurately models the interaction without congestion, Figure 5.20 includes the same minimal fit relative error using the experimental data from the *tds* environment. The *tds minimum* uses the same minimum fit machine parameters as the *minimum* which are shown in Figure 4.7. The assumption is that the minimal values derived from the experimental put results is the same on both ARCHER and the *tds* since they are the same hardware and by taking the minimal values any noise from congestion is filtered out.

The distinction between self-congestion and network congestion is also an important one to make. Self-congestion can never be addressed since it is an artefact from the *recursive multiplying* method itself, while network congestion can be avoided in a *tds* environment in which no other jobs are running. The relative error shown in Figure 5.20 shown as *tds minimum* contains effects of self-congestion, but not of net-

work congestion. When comparing the *minimum* and *tds minimum* we can see that the rising relative error is not present for the *tds minimum*. The conclusion is that the network congestion is primarily what is causing the relative error for the ARCHER environment.

5.5.6 Block-size Systematic Error Analysis

The block size in Section 5.5.2 was chosen such that the noise encountered from the measurements did not increase the maximum value, but also to mitigate effects of the finite timer resolution and the timing call overhead itself. Figure 5.21 shows resulting distributions of a 64 process AllReduce using the best schedule with varying number of samples per block of measurements. The red lines show the median (upper) and minimum (lower) of the block size of ten for easier comparison.

As seen in Figure 5.21 the minimums of all measurement block sizes are consistent. Therefore we have confidence the minimums were captured with the results presented in Figure 5.17. The medians show a slight upward trend correlated with block size. This causes the median calculated using a block size of ten to overestimate by approximately ten microseconds compared to the median using a block size of one. This overestimation is present for both the recursive doubling and the best recursive multiplying schedules.

5.5.7 Experimental-Model Correlation

As seen in Figure 5.20, the prediction using the model is not reliable for either the minimum or the median for the best schedule presented for the process counts. Using the model for prediction of the best schedule to use is not clear from the analysis. We plot all evaluated factored schedules in Figure 5.22 and Figure 5.23 using the experimental runtime value and the model prediction for that schedule. The number of processes is shown via the color of the points.

As shown in Figure 5.22 the schedule runtimes are not correctly predicted, but the clustering indicates that schedules are correctly identified as good or bad choices for a certain AllReduce size. Figure 5.23 shows that the median runtime is also incorrectly predicted, but a good schedule would also be chosen.

These results show that neither the minimum nor the median are predicted well by the theoretical model, but that a good schedule would be chosen regardless due to the clustering. This ill fitting is likely due to the effect that congestion has on the

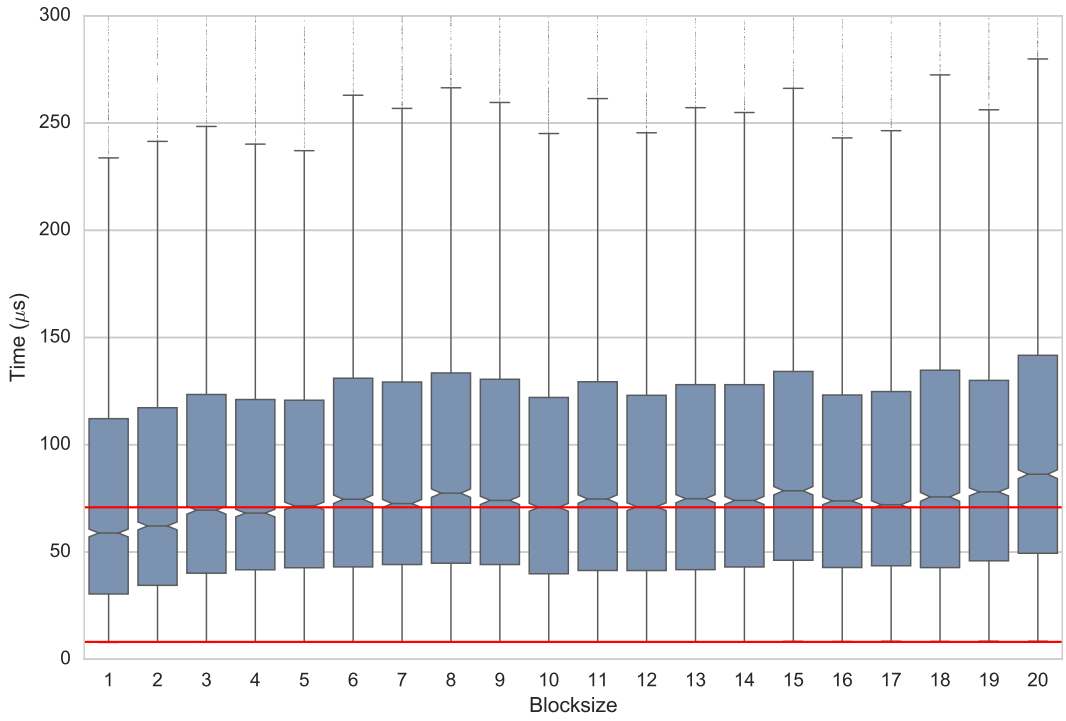


Figure 5.21: Distributions using different block sizes for the (a8, a8) schedule with 64 processes. The red lines show the median and minimum of the block size 10 distribution for comparison.

experimental results, which, as discussed in Section 5.5.3, is a large factor in the performance of the algorithms. The model does not attempt to include congestion effects and therefore fails to predict correct times. The model does approximately match the results shown in Figure 5.17 for the *tds* system.

5.5.8 Cray MPI Comparison

A direct comparison to the MPICH implementation of AllReduce is not representative due to the lack of an MPI library above our benchmark, but it is included to illustrate the benefits. Figure 5.24 shows the MPICH result alongside the recursive doubling and best recursive multiplying schedule. As shown, the minimums of the best schedule outperform both MPICH and the recursive doubling schedule as expected. This is true for the median values also.

From Figure 5.24 we can conclude that the Cray MPI implementation is indeed using the recursive doubling algorithm discussed in Section 5.2.2.2. In addition, we can see that the library overhead is negligible compared to the cost of the algorithm and congestion since the *mpich* and *rd* results are mostly identical. Finally, we can say that the recursive multiplying algorithm definitely outperforms the MPI_AllReduce implementation even if the library overhead is included.

5.6 Simulator Exploration

5.6.1 Factored Schedules

The recursive multiplying algorithm was previously explored both theoretically and experimentally. However, both failed to give a good representation of the actual execution. The theoretical diagram compresses the overlapping messages to a stage which causes skewing to be hidden, while the experimental measurements contain too much noise.

The examples shown in the following sections use a simulated pipelining latency-bandwidth model instance with the latency set to 500 nanoseconds, the pipeline latency to 100 nanoseconds and the bandwidth term set to 0.4 nanoseconds per byte. The compute tasks are set to take ten nanoseconds. The values for these parameters were chosen to illustrate interesting higher ratio effects, but also to be reasonably close to reality. Approximately a doubling of the maximal ratio from Table 4.1 was chosen. Since the local computation is a minor cost, it is not important. The algorithms are represented

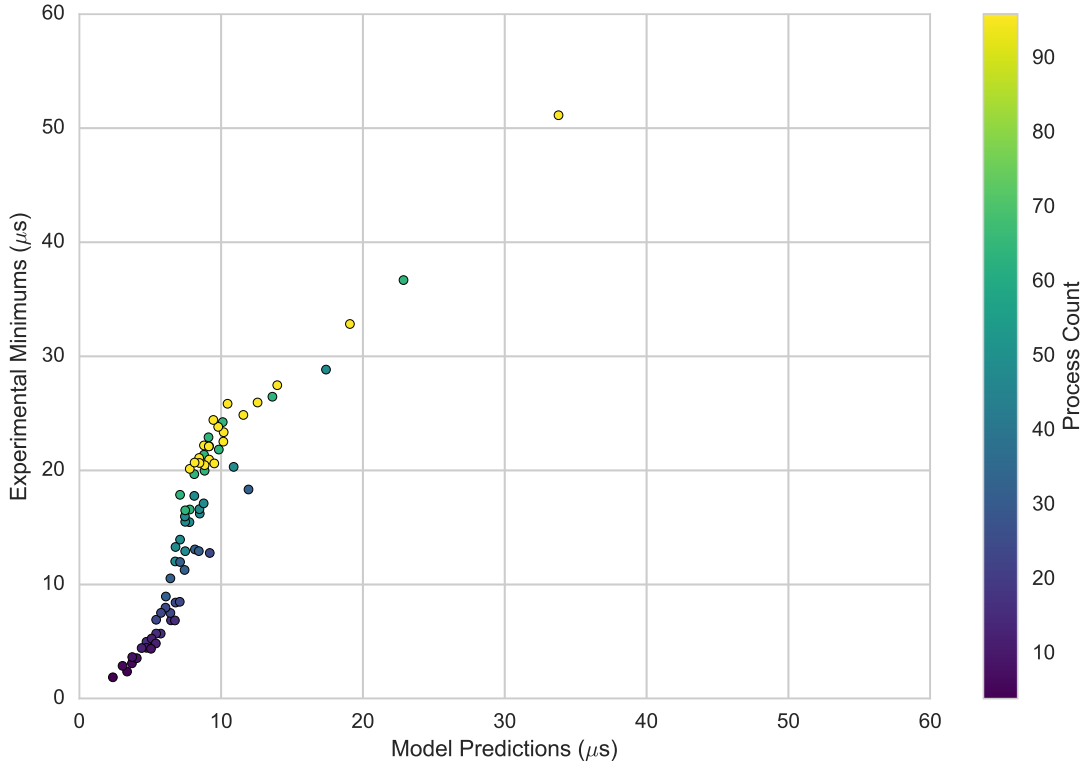


Figure 5.22: Correlation plot for experimental and model results using minimums.

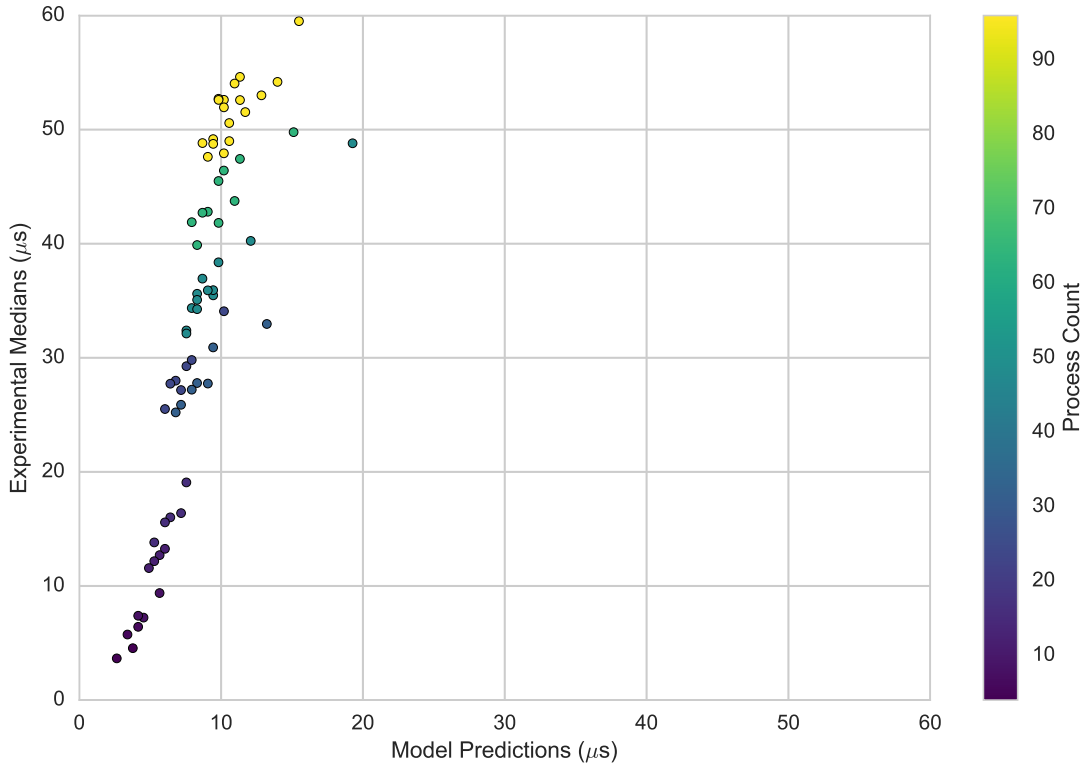


Figure 5.23: Correlation plot for experimental and model results using medians.

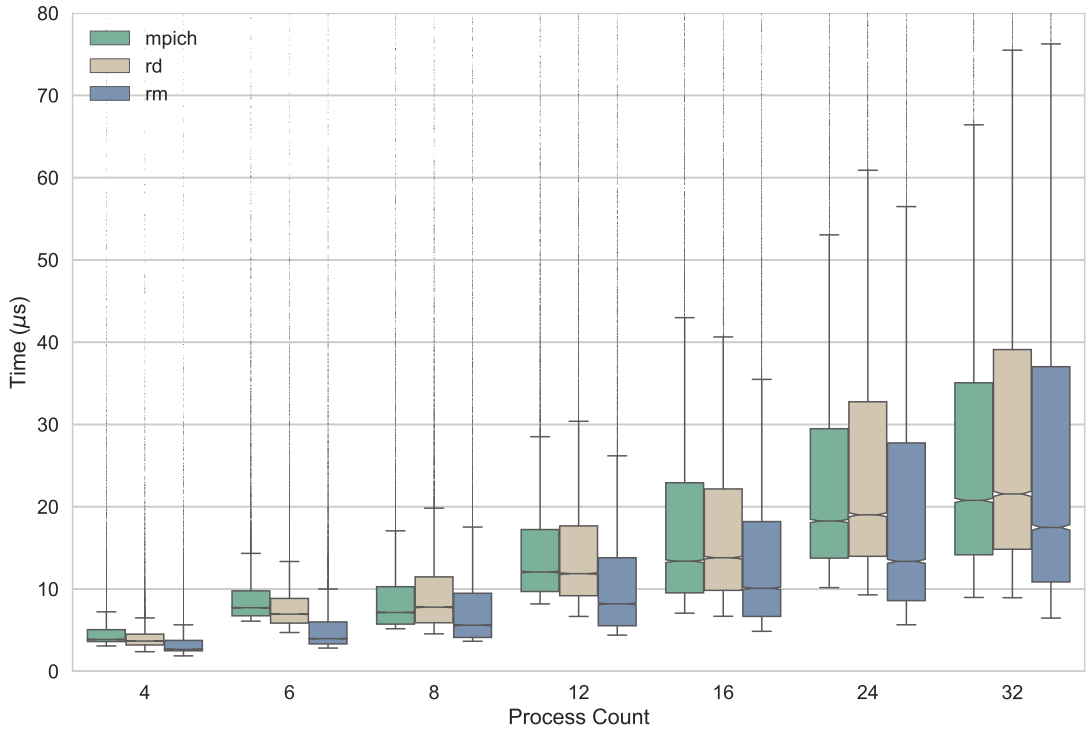


Figure 5.24: Execution times comparing directly the MPICH MPI_AllReduce implementation, the recursive doubling schedule as implemented in our benchmark and the best recursive multiplying schedule found.

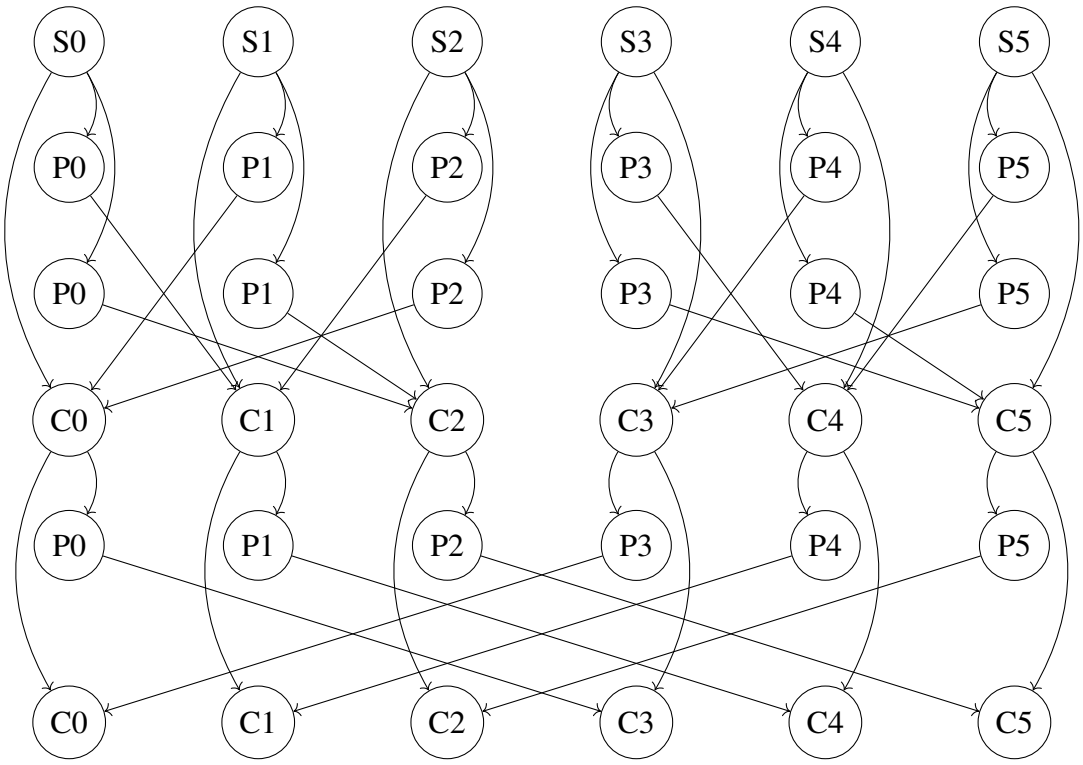


Figure 5.25: Program diagram of 6-way AllReduce using schedule (a3,a2). Start tasks are labelled with S, Put tasks are labelled with P and Compute tasks are labelled with C.

using the DAG program representation discussed in Chapter 4: an example is given in Figure 5.25 for a six-wide AllReduce using a *recursive multiplying* implementation.

Figure 5.26 shows execution of the recursive doubling schedule. As seen, overlapping is not occurring in this case since each sending process is waiting for the message to arrive on the receiving process. In comparison, Figure 5.27 shows the extreme case of all messages pipelining within a single stage. As shown previously, the message pipelining causes the multicast stage to be executed significantly faster.

5.6.2 Splitting & Merging

The recursive multiplying algorithm can evaluate AllReduce operations across many process counts. However, similarly to recursive doubling, some process counts are only possible inefficiently. For recursive multiplying, these are all prime numbers above a threshold determined by the overlap ratio discussed in Section 5.4.1.

We refer to the generalised collapse and expand method used in MPICH, and presented in Section 5.2.2.2, as a splitting method. Figure 5.28 shows the splitting method

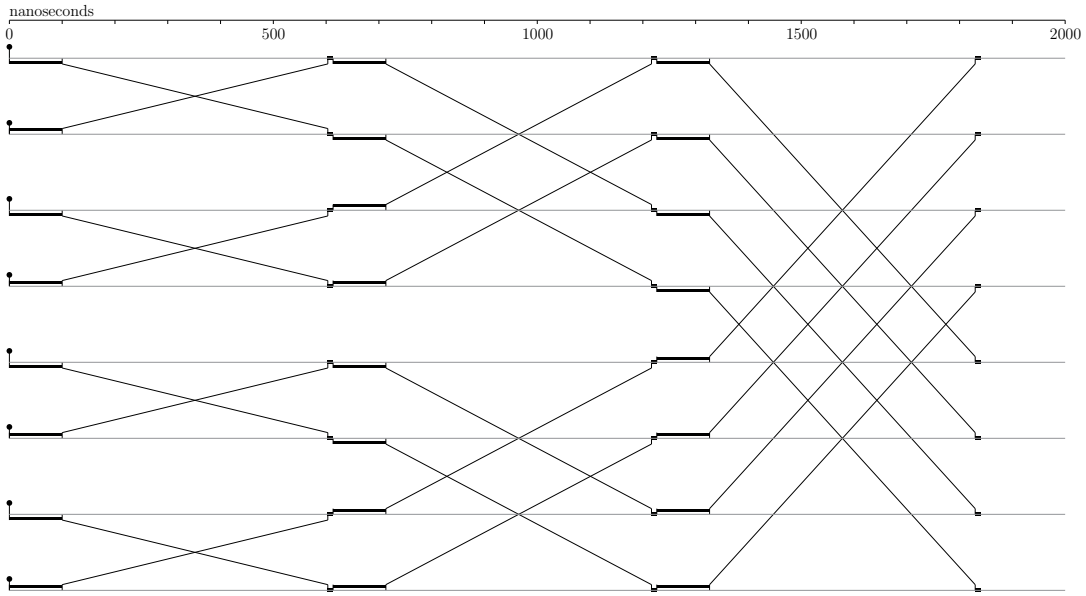


Figure 5.26: Simulated execution of the (a2, a2, a2) schedule AllReduce across eight processes. This schedule is the same as the recursive doubling algorithm.

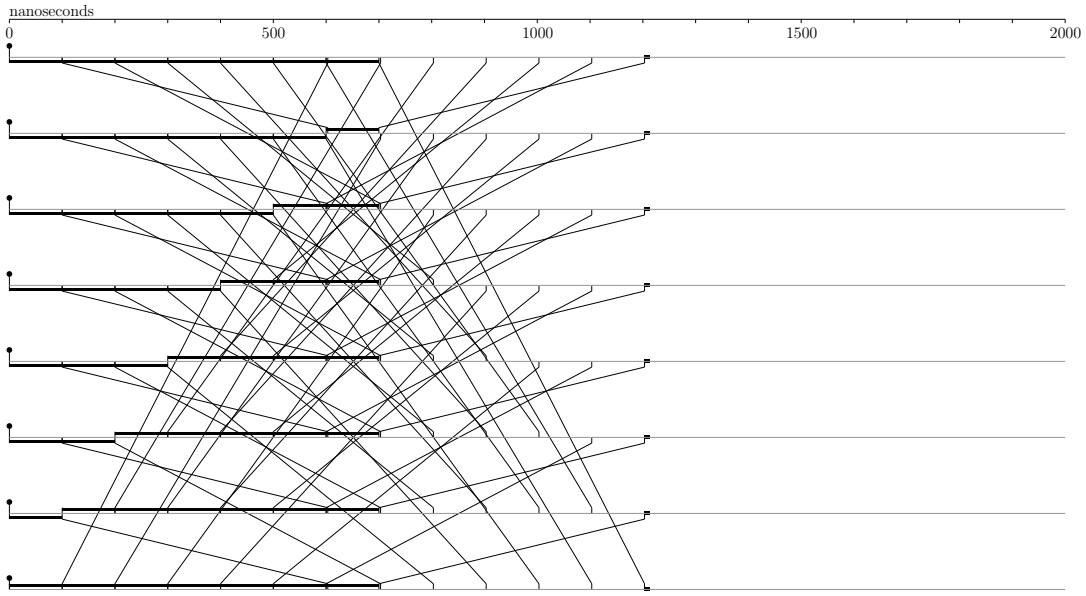


Figure 5.27: Simulated execution of the a8 schedule AllReduce operation across eight processes.

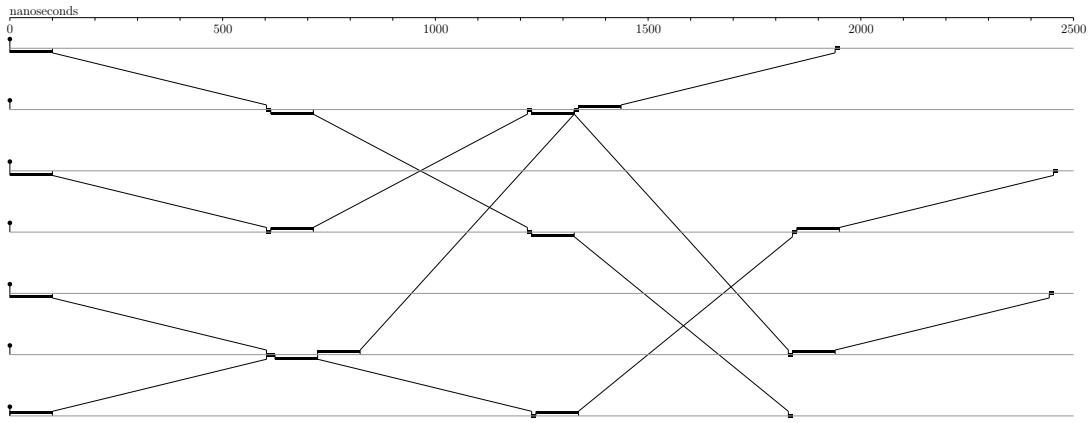


Figure 5.28: Simulator timeline of the splitting schedule (c6m2, a2, a2, e6m2) across seven processes.

applied to a $N = 7$ AllReduce. The interesting feature seen with the simulator is the skew which is introduced in an ideal execution of the algorithm. The collapsing processes send their data to their respective peers, as required, while the two internal peers are already executing stage two of the algorithm. This causes a skewed arrival at the second stage and subsequent stages. As seen, the finishing times for each process vary by approximately 700 nanoseconds. The specific skew introduced is dependent on the schedule chosen.

The merging method introduced in Section 5.4.2 is shown in Figures 5.29 and Figure 5.30. As seen in the figures, the runtime is decreased by approximately one microsecond (41.8%) compared to the recursive doubling schedule in Figure 5.28. Similar to the recursive doubling, skew is introduced. However, the skew is within 100 nanoseconds for the first schedule and 200 nanoseconds for the second schedule.

5.6.3 3-2 & 2-1 Elimination

Using the simulator, we are able to simulate not only the recursive multiplying algorithm, but also the 3-2 & 2-1 elimination method discussed by Rabenseifner and Träff [95]. Figure 5.31 shows an AllReduce operation executed using a 3-2 elimination. The simulator shows how the 3-2 elimination can overlap with in time with the pairwise exchange and thereby allow for the $\lceil \log_2 N \rceil$ bound. An $N = 7$ AllReduce is not a good test case for the elimination method, because it cannot be applied with any decomposition and therefore is nearly equivalent in runtime to recursive doubling in Figure 5.28.

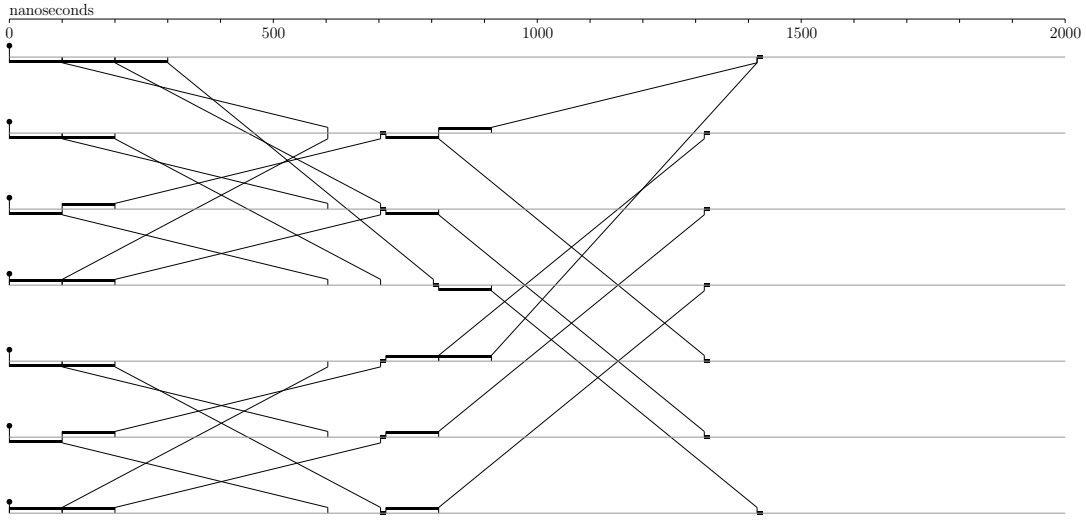


Figure 5.29: Simulator timeline of the merging schedule (m1g2a3, n1g3a2) across seven processes.

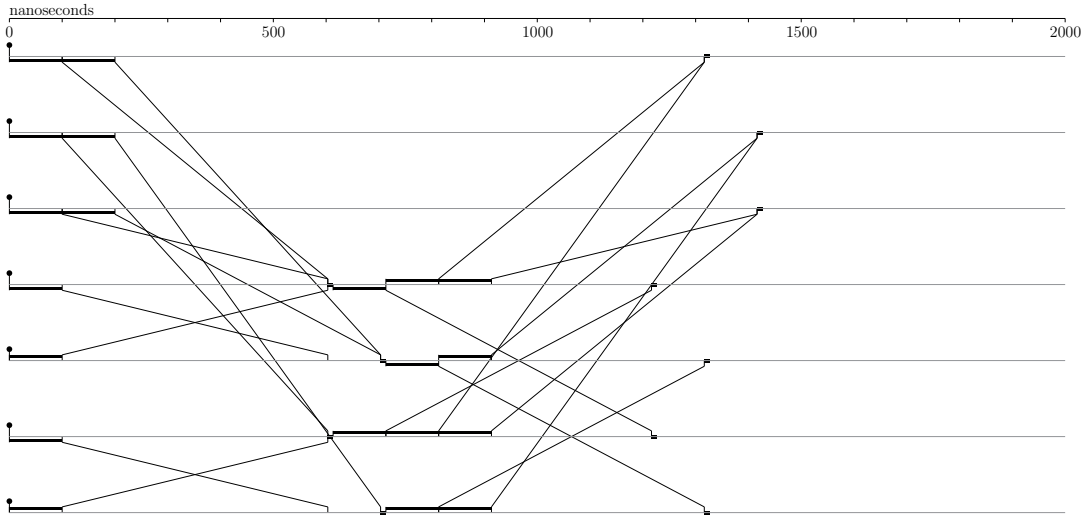


Figure 5.30: Simulator timeline of the merging schedule (m3g2a2, n3g2a2) across seven processes.

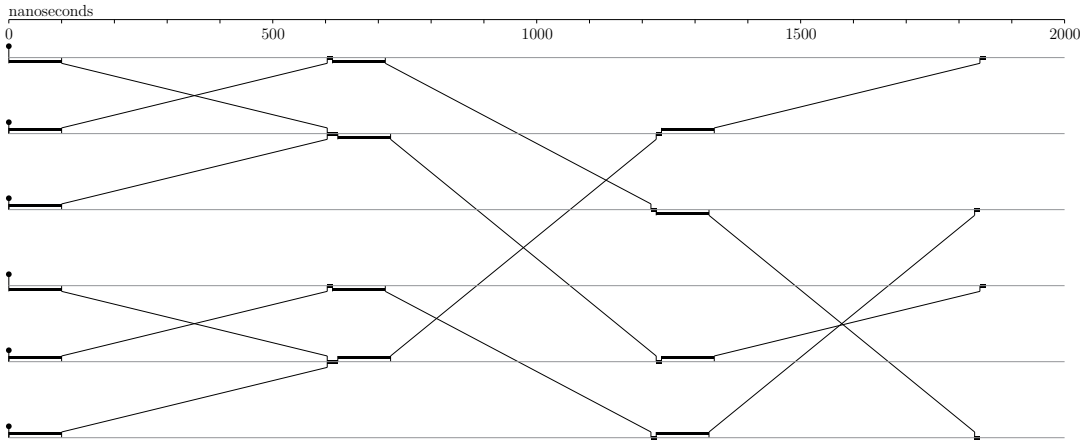


Figure 5.31: Simulator timeline of a non-overlapping elimination protocol with six processes.

For large non-power of two process counts the overall complexity bound of the elimination method is $O(\lceil \log_2 N \rceil + 1)$ for the latency optimized case. This is constructed with either the overlapping or non-overlapping method. The overlapping method uses 3-2 eliminations scheduled throughout the stages to achieve the bound in combination with the recursive doubling exchanges. The non-overlapping method eliminates all processes above the nearest power of two in the first stage using 3-2 and 2-1 eliminations (similar to the collapse/expand method), after which recursive doubling is applied. The larger the process count the closer the schedule acts to a normal recursive doubling schedule.

Chapter 6

Two-sided MPI Receive Queue

Minimising memory usage by MPI libraries, especially at scale, is of increasing concern because the memory per core for exascale supercomputers is predicted to be lower than for current petascale machines. We address the memory scalability of the software receive queue for point-to-point MPI communication by implementing a lockless, fixed-size, double-buffered queue capable of handling a single-consumer multiple-producer (*scmp*) usage pattern. Our new queue achieves constant memory usage for each MPI process, irrespective of the total number of MPI processes. We demonstrate improved memory usage for job sizes above 55 MPI processes, relative to the queue implementations from Cray MPI (a derivative of MPICH). Whilst the point-to-point latency using the new queue is larger, due to the shared state which requires synchronization, we assert that this will be an acceptable design trade-off at extreme scale. The scalability improvement gained by using the *scmp* algorithm is 10× compared to the previous lock-based implementation.

6.1 Introduction

High performance computing machines are growing larger and application sizes are increasing. To allow for this growth, the algorithms in all layers of the software stack must adjust to this new challenge. The Message Passing Interface(MPI)[85] contains point-to-point communication functionality that allows application developers to communicate between any pair of processes within a single communicator. Complex patterns of communication are not always easily described or efficiently implemented using the MPI collective operations, therefore point-to-point is provided as a flexible tool to permit any arbitrary communication pattern.

The definition of point-to-point operations in MPI requires temporary buffering of message headers that represent send operations until they are correctly matched with receive operations: this is done in matching queues. In addition to these, temporary space is required to receive the messages from the remote processes. This buffering is commonly implemented in modern MPI libraries using replicated queues, i.e. a set of queues at each process with one queue per communicating peer.

Historically, this design choice has been acceptable due to the abundance of available memory per process. However, with system and job sizes increasing towards exascale supercomputers and beyond, the memory per core is predicted to decrease significantly, perhaps by orders of magnitude[29, 43]. The memory that is currently used for queue replication will be required by user applications, instead of the communications library.

The EMPI4Re[33] MPI library is a research vehicle for investigating and prototyping new MPI semantics and implementation ideas. The point-to-point implementation uses a single message buffer queue per process, which aims to achieve a constant memory footprint implementation of MPI. However, this single queue only supports single-consumer single-producer usage and therefore does not scale well, due to congestion and synchronisation required by a many-to-one communication pattern. Our work improves the scalability without losing the constant memory footprint, which supports operating in a memory-limited environment such as predicted for exascale supercomputers.

To remedy the scalability issues present in the EMPI4Re library, we introduce a new queue mechanism based on lockless queues using remote atomic operations present in the Cray Aries NIC[35]. This allows for a message insertion protocol with minimal congestion effects. We compare the performance and memory usage to the

Cray MPI implementation, which is the default installed MPI library on Cray platforms.

6.1.1 Contributions

- A shared-memory lockless single-consumer multi-producer queue, intended for remote queue insertion, is introduced.
- The SCMP queue achieves lower latency and has much better congestion capabilities compared to locked queues for a memory-limited setting.

6.1.2 Overview

Section 6.2 presents the queue methods used originally in EMPI4Re and Cray MPI. Section 6.3 introduces the lockless single-consumer multiple-producer queue algorithm now used in EMPI4Re. Section 6.4 presents experimental evidence for the viability and performance of the new algorithm.

6.2 Prior Work

6.2.1 Cray MPI

The Cray MPI library is a derivative of the MPICH[47] implementation of the MPI standard. The MPICH library is widely used as the base for vendor implementations of MPI. The Netmod interface[91, 92] allows high performance to be achieved on different network platforms without rewriting the entire library.

The Cray MPI implementation contains two queue implementations that are used with small messages for eager message and protocol message transmission. These are the *smsg* queue and the *msgq* queue, which can be chosen by using an environment variable. Both these queues operate in a SCSP – single-consumer single-producer – fashion. At the time of writing, there are no multi-producer queues in Cray MPI. The queue memory can be allocated either at startup, or dynamically as processes initially communicate.

The *smsg* queue aims at achieving minimal latency between two processes. However, it requires a mailbox per remote process on every process. The queue is likely implemented as a circular buffer using two integer fields as pointers into the mailbox buffer, stored on both the consumer and producer processes. To insert an eager message

without participation from the consumer process only requires two `put` operations; one containing both the message header and the message data, and another to update the value of the tail pointer at the remote process. This operation can be combined into a single network operation using a put and flag message.

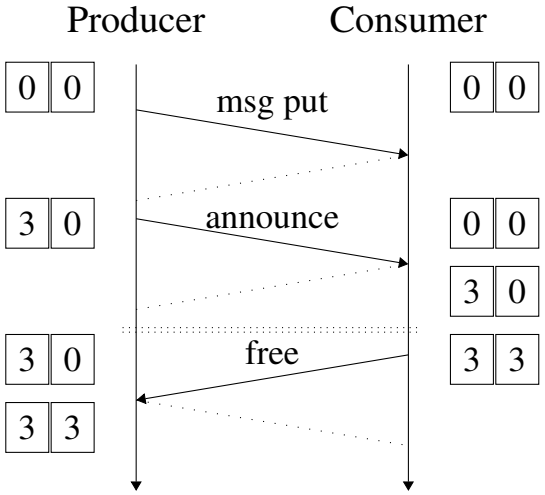


Figure 6.1: Illustration of the *smsg* queue mechanism present in Cray MPI.

The *smsg* queue protocol is illustrated in Figure 6.1. Given a message size below the eager message threshold, this algorithm is executed by the producer. The producer can (independently of the consumer) determine if enough space is left in the queue for the message, due to the mirrored local counters, as shown in Figure 6.1.

When the consumer process removes messages from the queue on the remote side, it thereby releases memory for more messages to be sent. The consumer is responsible for updating the head pointer, both locally and in the mirrored counters at the producer process. This allows for latency hiding and therefore is not on the critical path of the message insert. Latency hiding could also be achieved by issuing a non-blocking get operation at the beginning of the protocol from the producer, to fetch the tail counter value from the consumer.

The *msgq* queue mechanism reduces the number of queues by sharing each one between all processes on a single node. This reduces the memory overhead of *msgq*, relative to *smsg*, but slightly increases communication latency. A single process per node is assigned as the 'leader' process. The leader process executes all network communications with the receiving process, using the same circular buffer algorithm as is used with the *smsg* queue mechanism. A local sharing mechanism, using shared memory operations within each node, allows access to the message queues by other local MPI processes.

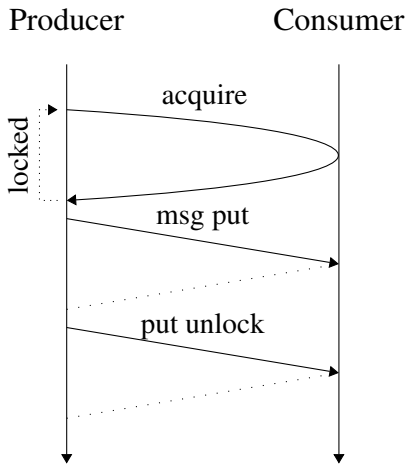


Figure 6.2: Illustration of lock-based queue mechanism communications used in EMPI4Re.

6.2.2 EMPI4Re

The EMPI4Re MPI implementation is largely based on the T3D library[16] designed for MPI 1.0. The library is entirely based on remote direct memory access operations, which allow processes to modify each others' memory. The goal of EMPI4Re is to permit research and rapid prototyping of novel MPI concepts, semantics, and implementation choices. Although this work leveraged EMPI4Re for benchmark performance testing, the new *scmp* queue implementation can be applied to other MPI libraries.

There are several protocols for point-to-point functionality in the EMPI4Re library. The 'T1' protocol is an eager message protocol aimed at transmitting a small message as soon as possible. It places a single protocol packet, containing both the message header and the message data, into the protocol queue at a remote process.

The 'Tn' protocol is an eager message protocol for larger messages. It also uses the protocol queue for both message header and message data, but it transfers several protocol packets at once. The first of these protocol packets contains the message header: the others contain the message data.

The 'RTA' and 'RAT' protocols are rendezvous message protocols aimed at achieving maximum bandwidth for very large messages, They require a full network round-trip and multiple protocol packets, which use the protocol queue, to negotiate the transfer of message data directly between the user's send and receive buffers.

The protocol queue is a lock-based double-buffered queue with fixed-size slots. This supports the many-to-one communication required for MPI applications. Fig-

Algorithm 8 Producer-side lock-based algorithm.

▷ acquire lock

1: **repeat**
2: index ← **afor** lock
3: **until** lock acquired

▷ transfer messages and unlock

4: **put flag** count messages to Q and unlock

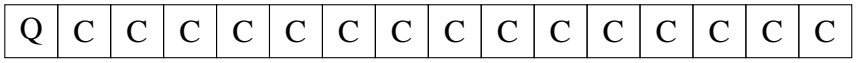


Figure 6.3: Illustration of lock encoding using individual bits for the queue and offset into the queue.

ure 6.2 shows a 'T1' protocol message insertion from a producer process to a consumer process.

The first part of the algorithm is to acquire a remote lock using a remote atomic operation. The local process can then safely `put` a message into the queue at the remote process. The remote queue is then unlocked with a `put` to the lock memory location on the remote process. Algorithm 8 shows the pseudocode used by the sending process for locking the remote queue. Algorithm 9 shows the pseudocode used by the receiving process to empty the protocol queue and progress the application.

Algorithm 9 Consumer-side lock-based algorithm.

▷ test for work

1: **if** Q is non-empty **then**

▷ acquire lock

2: **repeat**
3: index ← **afor** lock
4: **until** lock acquired

▷ release other queue

5: **put** locked swap queue

▷ process received messages

6: process(Q)

The pseudocode presented in Algorithm 8 and Algorithm 9 uses a bitwise encoding to transmit both the state of the lock and the offset into the queue with a single network operation. Figure 6.3 shows the bit pattern of the lock variable. Using an *afor*

instruction – an atomic fetch and bitwise OR – allows a lock to be set if it is unset on the most significant bit.

If the lock is already acquired by another process, the atomic OR operation will not change the state of the lock. The process trying to insert a message will need to retry until the lock is acquired. When the lock is acquired and the state is returned to the sending process, the offset can be calculated by using the appropriate mask with a bitwise *and* instruction operation.

6.2.3 Related Work

The software queues implemented in Cray MPI and EMPI4Re are well understood queues for shared memory in a multi-threaded environment. Shared-memory queues have been a subject of research for a long time [44, 73, 81, 115]. The usage of these queues in a network context is the differentiating factor for our work, since the network communication is orders of magnitude longer than any shared-memory transaction. The avoidance of network communications in addition to congestion effects are vital for a good implementation in such situations.

An alternative to software receive queues is to use hardware receive queues. The Cray T3E implemented a queue insert operation in the NIC [10], which allowed atomic insertion of messages to the remote process instead of a message protocol requiring a form of locking. A modern usage of more capable hardware is used in Portals 4 [8] to directly insert messages into the matching queues, instead of using software receive queues that are then processed into an appropriate data structure.

6.3 SCMP Algorithm

The double-buffered lock-based queue, discussed in Section 6.2.2, has a congestion problem at the receiving side. The goal of the new queue mechanism is to allow for a large, many-to-one, insert pattern at the receiver with minimal network communication. This enables the implementation of single-threaded MPI with a single queue buffer, instead of duplicating the queue buffer for each remote process, and thereby allows a constant memory implementation.

The unique characteristic of our method is the use of the *afax* instruction, available on the Cray Aries, used to construct a minimally conflicting single-consumer many-producer queue. This queue mechanism can only be used on networks which support

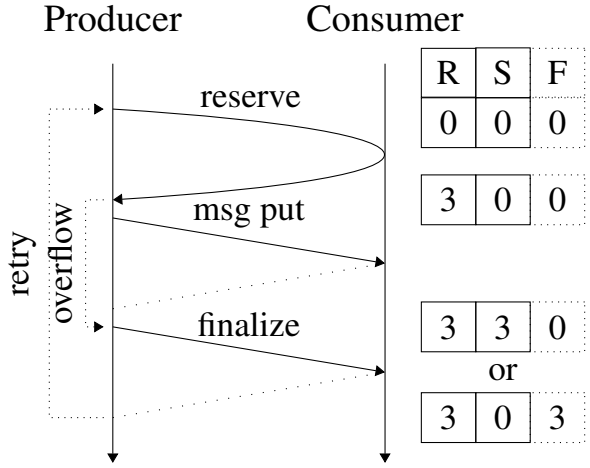


Figure 6.4: Illustration of the scmp queue mechanism implemented in EMPI4Re.

the *afax* instruction of which there are no other examples as we know. Libfabric[48] supports multiple interesting atomic network operations, but the *afax* instruction instruction is not among them. It may be possible to construct a similar locking behaviour by use of the *mswap* operation.

The main concern of the new queue algorithm is a reduction in the impact of the congestion that is present in a many-to-one pattern. A reduction in the potential for congestion can be achieved by reducing the amount of time during which multiple remote processes can interfere with each other. The queue data structure will nonetheless require shared state to be retrieved before an insert can be executed. This choice is made in respect to the memory to latency trade-off. The double-buffered structure has been carried over, since this reduces the contention between consumer and producers.

In the existing *rlock* queue algorithm in EMPI4Re, the entire remote queue is locked during each insertion operation by a remote process. In the new *scmp* queue algorithm, only the section of the queue required by the remote process is locked. This permits greater concurrency because multiple remote processes can safely issue `put` instructions to non-overlapping sections of the queue. Ensuring that attempts to reserve a subset of the queue are always successful guarantees that remote peers will only busy-wait over the network if an overflow occurs. In the *scmp* queue algorithm, a failure to lock a requested section of the queue can only happen if the queue is temporarily out of space.

Figure 6.4 shows the communication and control flow executed by a producer to a consumer process. The `get` operation, shown as *reserve* in the diagram, is the only network operation that can cause congestion with other processes in the system. This

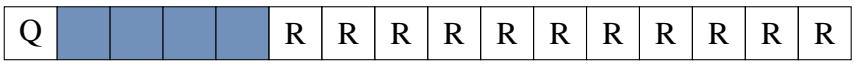


Figure 6.5: Illustration of *R* counter encoding using individual bits for the queue and offset into the queue.

operation retrieves the counter state from the consumer. If the counters indicate that an overflow would occur due to its insert operation, then the producer raises a fatal error. Otherwise, the producer inserts its messages into the reserved section of the remote queue (shown as *msg put* in the diagram). Once the transfer is complete, the producer informs the consumer that the operation is complete (shown as *finalize* in the diagram).

If the counters retrieved from the consumer indicate that an overflow would occur, then the data transfer is skipped but the *finalize* stage must be executed to invalidate the reservation and allow the consumer to process the queue. The producer can retry, or choose a different way of handling the exception. The overflow exception will occur when the message from the sending process is too large for the remaining space in the queue.

The *R* counter is an encoded integer that indicates which of the two parts of the double-buffered queue to use and the position within the queue. Figure 6.5 shows an example of the encoding using a 16-bit integer. The leftmost bit of the *R* counter determines which half of the queue is currently being used for insertions. A number of spare bits are left empty and the offset value is stored in the rightmost bits. The *S* counter and *F* counter are the successful and failed slot counters respectively. These are used by producers to indicate whether their reservations were executed successfully or failed.

The spare bits allow for an overflow of the offset value without corrupting the queue bit. The number of spare bits limits the total size of failing reservation requests. When a producer process requests more space than is currently available, then the offset value will overflow into the spare bits. Further requests, occurring before that process releases its failed reservation, will cause the offset value to overflow even more. If the total size of these failing reservations requests becomes too big, then the leftmost bit will be corrupted. This is a fatal error for this queuing algorithm. Using a 64-bit integer allows for a sufficiently large total size of failing reservations.

Using a 64-bit integer, the leftmost bit will be required for the queue index, then a variable number of spare bits can be allocated while the remaining bits are used as the offset into the queue. If the queue length required is 10^6 slots, then only 20 bits are

Algorithm 10 Producer-side scmp queue algorithm.

```
                                                  ▷ reserve remote queue space
1: announce state ← afadd count
                                                  ▷ select pointers
2: Q, R, S, F ← select(announce state)
                                                  ▷ check for overflow
3: if R + count > queue size then
                                                  ▷ finalize failed transfer
4:     F ← aadd count
5: else                                           ▷ transfer messages
6:     put count messages to Q
                                                  ▷ finalize successful transfer
7:     S ← aadd count
```

required for the position value. This leaves 43 spare bits which would be practically impossible to overflow.

Algorithm 10 shows pseudocode for the producer-side of the *scmp* queue implementation, shown in Figure 6.4. The *afadd* instruction is an atomic fetching add, which allows a one-way reservation to be added to the offset bits within the encoded *R* counter. This locks a section of the queue starting at the original offset and of the right length to insert all the messages from the producer. If the end of the reserved section is beyond the size of the queue, the queue will overflow with the additional messages, so the reservation is released immediately, via an *aadd* instruction operation using the *F* counter. Otherwise, the producer writes data into the reserved section of the remote queue and releases the reservation via an *aadd* instruction operation using the *S* counter.

Algorithm 11 shows pseudocode for the consumer-side of the *scmp* queue implementation. The *afax* instruction instruction swaps the double-buffered queue, by toggling the leftmost bit and setting the rest of the counter to zero. The atomic instruction also returns the *R* counter state in a single atomic operation. This instruction is present on the Cray Aries NIC. Once the correct counters are selected, the consumer waits until all remote peers have released their reservations. Next, the successfully transferred messages are removed from the queue and processed according the normal MPI matching rules. When a slot is emptied, the message header is zeroed. Finally, the *S* counter and *F* counter are reset to zero.

Algorithm 11 Consumer-side scmp queue algorithm.

▷ fetch current state and swap queues

1: announce state ← **afax** queue mask

▷ select pointers

2: Q, R, S, F ← select(announce state)

▷ wait for remote peers

3: **while** F+S ≠ R **do** wait

▷ process received messages

4: process(Q, S)

▷ reset counters

5: S, F **put** 0

The *afax* instruction is a requirement for the consumer-side, because the queue cannot be locked in a traditional way. The *afax* instruction is a combined atomic AND and XOR operation, which we use to zero the offset encoded in the *R* counter and to flip the queue bit.

6.4 Experiments

6.4.1 Environment

All experiments presented in this section were run on ARCHER [32], a Cray XC30 machine with 4920 compute nodes, each with two 12-core Intel E5-2697v2 CPUs. The interconnect is the Cray Aries in a Dragonfly topology. The environment in which the experiments were executed is, except where mentioned:

- PrgEnv-cray/5.2.56
- craype/2.4.2
- cray-mpich/7.2.6
- dmapp/7.0.1-1.0502.10246.8.47.ari
- pmi/5.0.7-1.0000.10678.155.25.ari
- ugni/6.0-1.0502.10245.9.9.ari

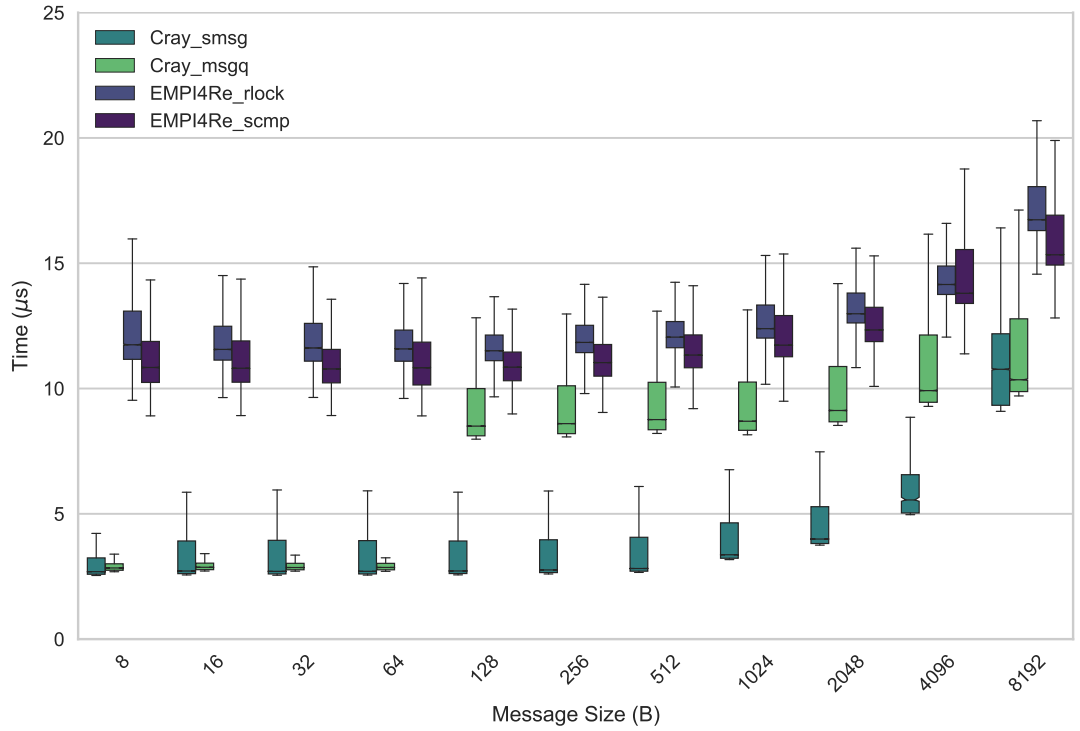


Figure 6.6: Latency distribution with message size collected using the *smsg*, *msgq* and *scmp* queue implementations. All message sizes are powers of two, boxplots are offset for clarity.

The default queue settings were used in all cases for the Cray MPI tests. For the *scmp* and *rlock* algorithms, the queue length was 1024 for each half of the double buffered queue.

6.4.2 Latency

The latency properties of the queue algorithms are an important aspect of the costs to consider when determining which to use. Figure 6.6 shows the latencies measured on ARCHER. The ping pong benchmark used consisted of ten ping pongs in a block, over which a mean was calculated to reduce the impact of high frequency noise. The ping pong benchmark was implemented using the MPI_Send function with the default environment flags for thresholds. The number of samples per queue algorithm was 143000; this number is large enough to ensure a stable distribution with a live environment.

Figure 6.6 shows the behaviour of all queue algorithms with respect to message size. The eager message limit for Cray MPI is set at 8KB, therefore the last message size shown in Figure 6.6 is using a rendezvous message path. Due to a limitation

within uGNI [91, 92], *msgq* uses a different message transfer region for message sizes at and above 128 bytes. This also causes a rendezvous path to be used for message transmission, which can be clearly seen as a jump from, approximately, 3 μs to 9 μs. As can be seen, the *scmp* protocol length has caused the latency to be about 3.5$\times$ the latency of *smsg* at 8B.

6.4.3 Memory Scaling

The *scmp* algorithm was designed to use only a fixed-size buffer for receiving messages, while the *smsg* and *msgq* algorithms both use queue replication. The benchmark used to measure the scaling behaviour of memory usage was a naive all-to-all communication pattern. The measurement of memory usage was done through the POSIX standard `getrusage` function. To calculate the difference in memory required for the communication pattern, the memory was read before and after the communications phase. However, this does not allow us to separate the matching queue memory from the receive queue memory.

The use of an all-to-all communication pattern is the best case for our *scmp* queue mechanism compared to the *smsg* and *msgq* mechanisms, which are optimized for lower number of communication peers. For communication graphs which contain nodes with lower degrees it may be optimal to use either of the previously implemented methods. In other words, the *scmp* method provides greater choice to library implementors to provide a balance between memory consumption, latency, and bandwidth. Combining the *scmp* queue mechanism with hashed receive queues[100] allows for even more flexibility. The optimization between all queue mechanisms is dependent on the specific needs of applications and the hardware provided.

Figure 6.7 shows the memory usage for all queue algorithms discussed. The *smsg* algorithm clearly uses the most memory: second is the *msgq* algorithm, which uses significantly less. The discontinuity seen for both the *smsg* and *msgq* are likely due to a block allocation of more mailboxes for receiving or for the matching queues. However, this cannot be verified since it is closed source. With *scmp* and *rlock* these are allocations of additional matching queue items. Similarly to *smsg* queue, the *msgq* queue allocates mailboxes in blocks. However, due to the lower rate of usage (per node rather per process), the allocation of a further block is delayed - until 1024 processes in this plot.

The *rlock* and *scmp* algorithms used in EMPI4Re are the most memory efficient

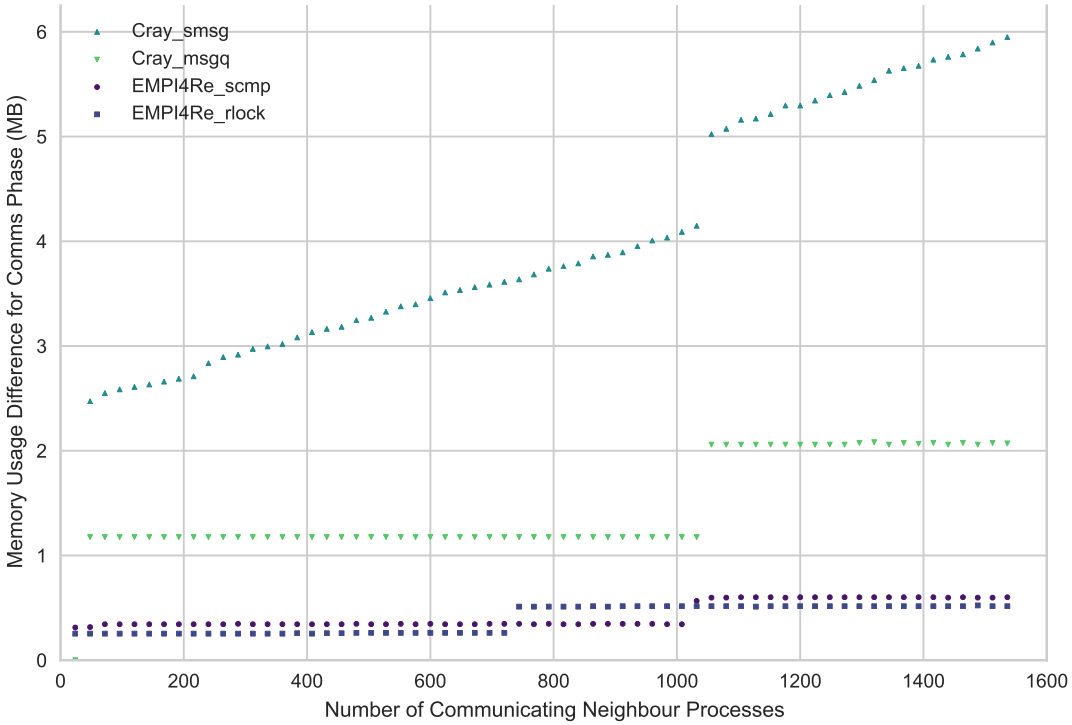


Figure 6.7: Maximum virtual RAM usage with all-to-all communication pattern.

algorithms tested, since they do not allocate additional memory for receive queues. The increase in memory usage for the *rlock* and *scmp* algorithms is due to increasing the size of the `PtEvNotice` pool, which permits a longer matching list. The difference in neighbour count at which the jump happens is due to the difference in contention between the *scmp* and *rlock* queue implementations directly affecting the achievable message matching rate, and therefore the high-water mark match list length required. Preallocating a greater number of `PtEvNotice` structures during initialisation removes the jump, by increasing memory usage for the tests with fewer communicating neighbours.

An analytic plot of memory consumption per process for each queue algorithm is shown in Figure 6.8. These results are found using the predicted memory consumption given in Pritchard et al.[91], but they do not account for block allocation of the queues as implemented in Cray MPI. As seen, the modelled memory usage roughly matches the experimental results given in Figure 6.7, but without accounting for the matching queue memory or other preallocated buffers.

For ARCHER, these results show a benefit in memory consumption after 56 processes compared to the *smsg* algorithm and 73 processes compared to the *msgq* algorithm. These process counts are calculated using Equation 6.1 and Equation 6.2 with

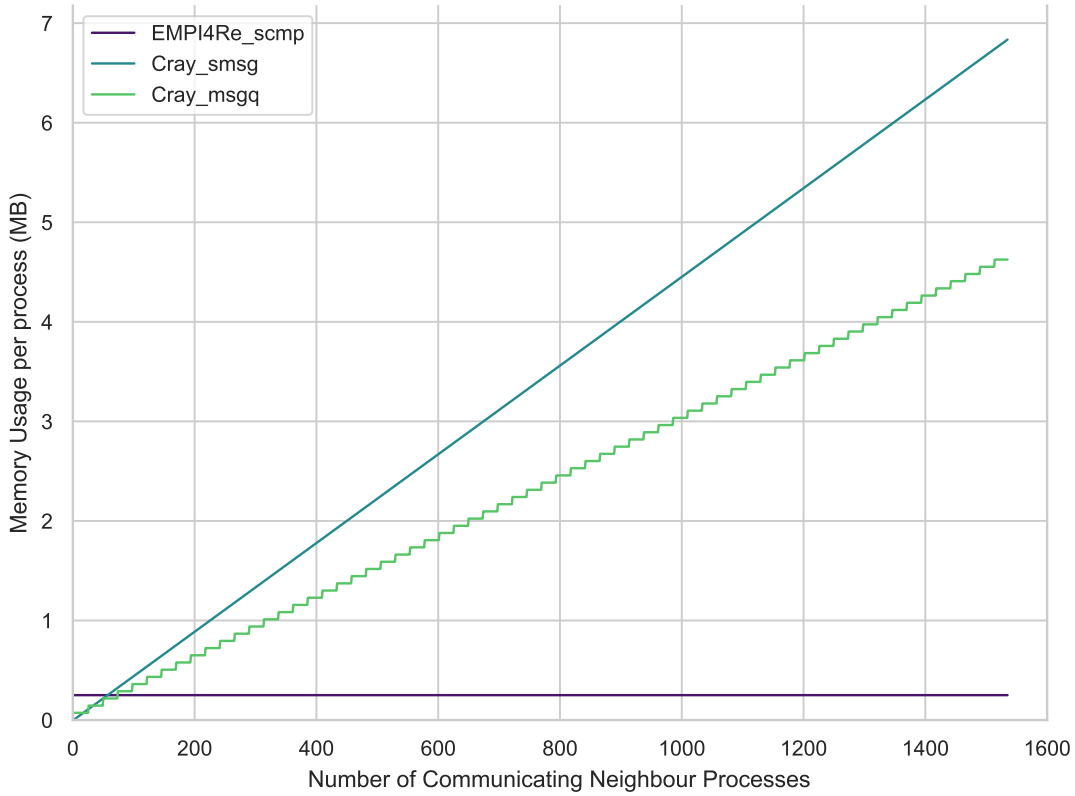


Figure 6.8: Analytically modelled total memory usage per process for an all-to-all communication pattern with various queue algorithms. We assume 24 processes per communication node.

the default values for ARCHER. The *rlock* algorithm is not shown since it is equivalent in memory consumption to the *scmp* algorithm. The *msgq* measurements were completed with the cray-mpich/7.5.2 module compared to the default environment, due to a software bug.

$$\text{threshold}_{\text{SMSG}} = \frac{\text{buffer\_size}_{\text{SCMP}}}{\text{usage\_per\_process}_{\text{SMSG}}} \tag{6.1}$$

$$\text{threshold}_{\text{MSGQ}} = \left\lfloor \frac{\text{buffer\_size}_{\text{SCMP}}}{\text{usage\_per\_node}_{\text{MSGQ}}} \right\rfloor \times \text{processes\_per\_node} + 1 \tag{6.2}$$

Equation 6.1 and Equation 6.2 can be used to calculate the threshold process count after which memory is saved by using the *scmp* mechanism. The tunable parameters are not restricted to any specific system or network. These equations only optimize for memory usage and not for overall performance. Overall performance would be a function of memory usage, latency requirements and other factors.

6.4.4 Temporal Scaling

The design of the *scmp* queue algorithm is geared towards allowing many peers to insert messages without congesting the queue state. Therefore, a simple test using ping pong is no longer meaningful, because the limiting factor becomes the response rate of the receiver compared to the receiver's ability to receive the ping messages.

Since we know that both the MPICH and EMPI4Re libraries will use an eager message transmission below their respective thresholds, it is possible to force these libraries to congest at the receiver end. To benchmark temporal scaling of the queue algorithms, we set up a many-to-one communication pattern in which each peer sends eight messages below the eager threshold to the receiving root process. This forces all peers to use the eager queue algorithms and causes a sufficient amount of congestion. Each queue algorithm was sampled 24500 times; the sample count was chosen such that the resulting distributions were stable.

Figure 6.9 shows the scaling behaviour of *rlock* when multiple active peers attempt to write at approximately the same time. Increasing the number of active peers has a large effect on time of completion of the sender processes. Figure 6.10 shows the equivalent experiment using the *scmp* queue algorithm. The impact of congestion on the performance of the *scmp* algorithm is significantly less. For comparison, Figure 6.11 and Figure 6.12 show the *smsg* and *msgq* algorithms. Comparing these with the *scmp* algorithm, the *scmp* algorithm performs equivalently in terms of scalability, but it is worse in terms of absolute latency, as expected due to the critical path length of the protocol. The scalability is improved by approximately $10\times$ compared to the previous *rlock* implementation.

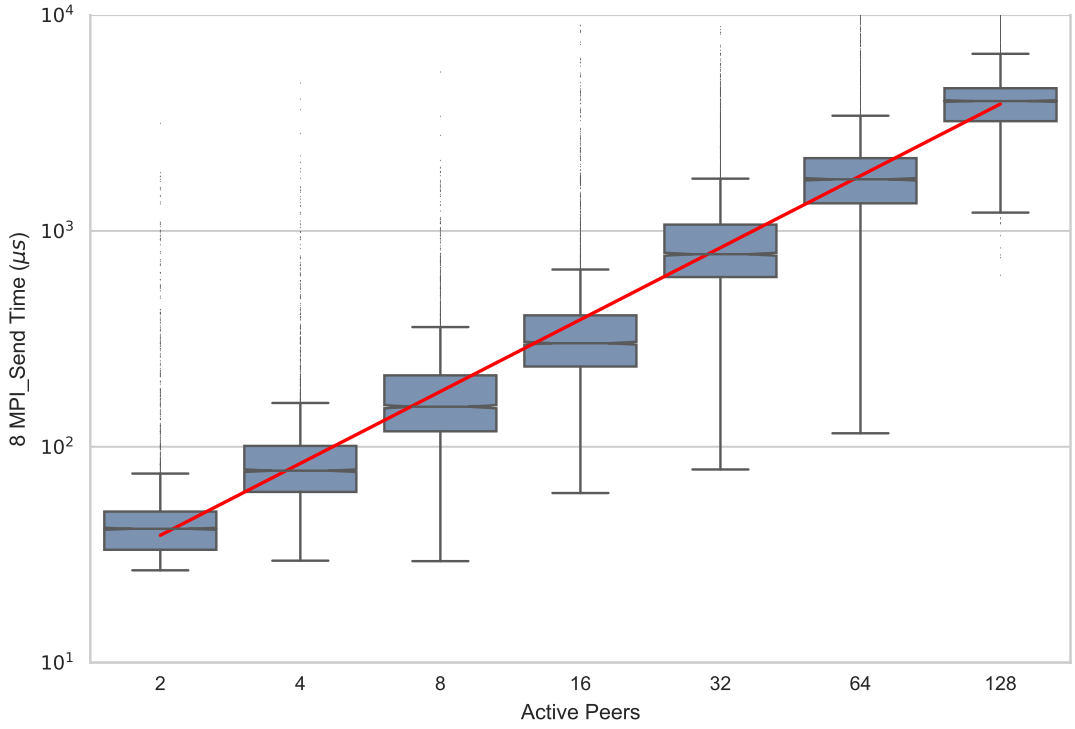


Figure 6.9: Distributions of insertion times using the *rlock* queue algorithm. The red line shows the linear regression through the 99 percentile data.

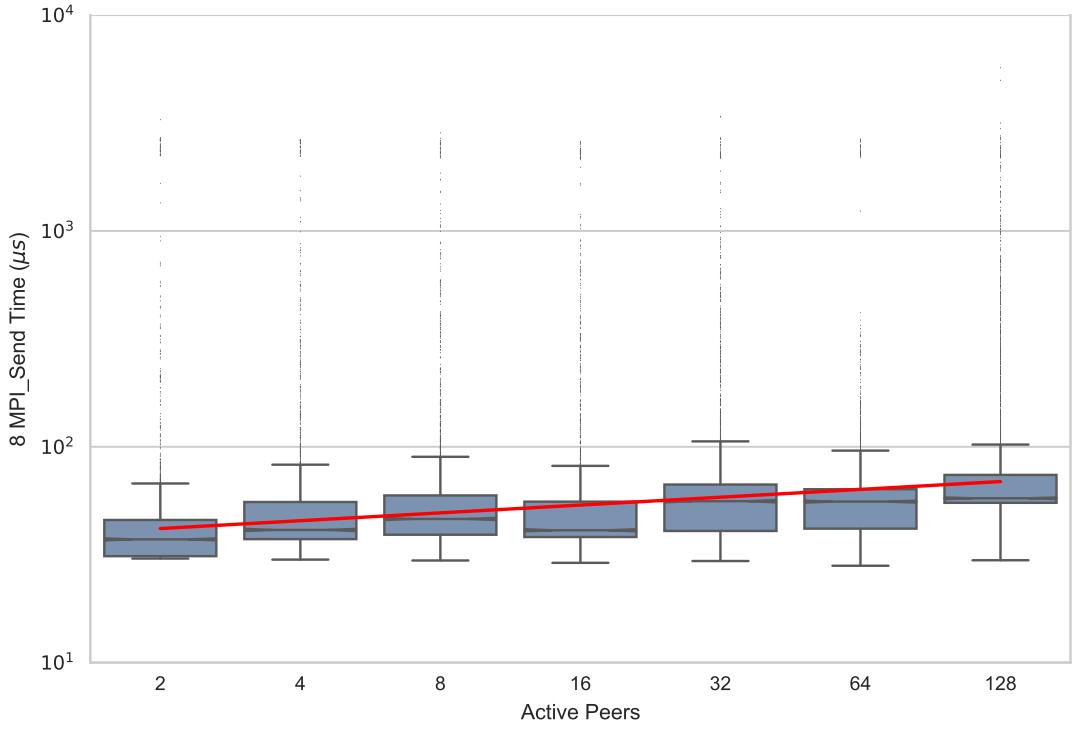


Figure 6.10: Distributions of insertion times using the *scmp* queue algorithm. The red line shows the linear regression through the 99 percentile data.

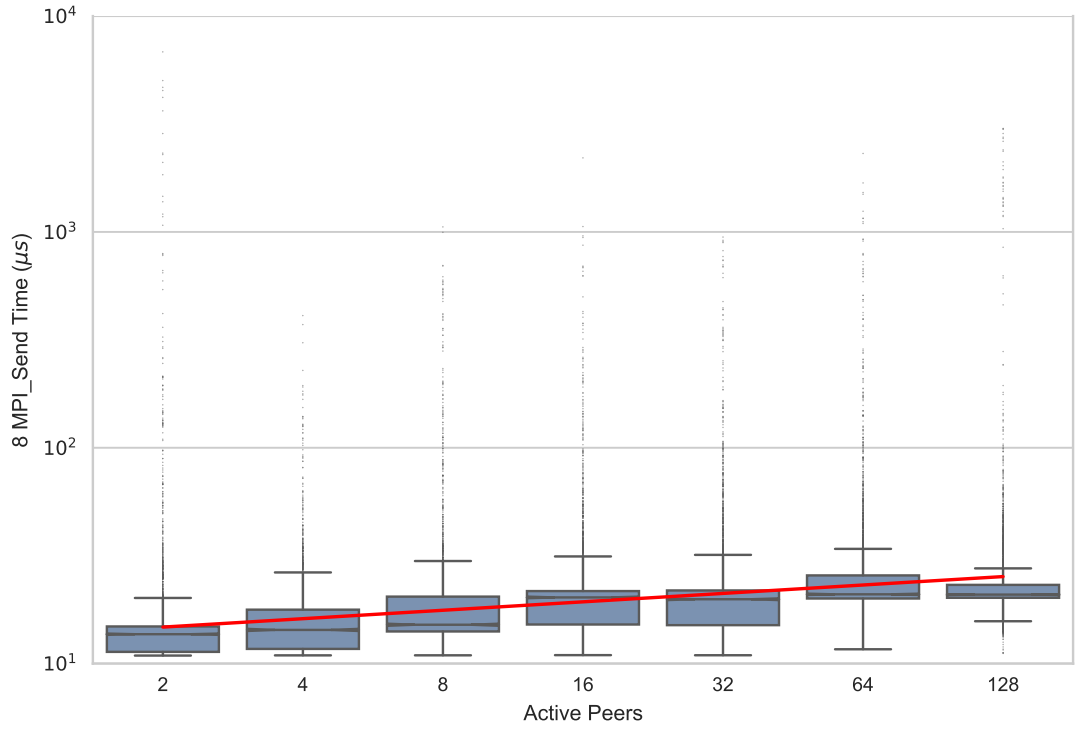


Figure 6.11: Distributions of insertion times using the *smsg* queue algorithm. The red line shows the linear regression through the 99 percentile data.

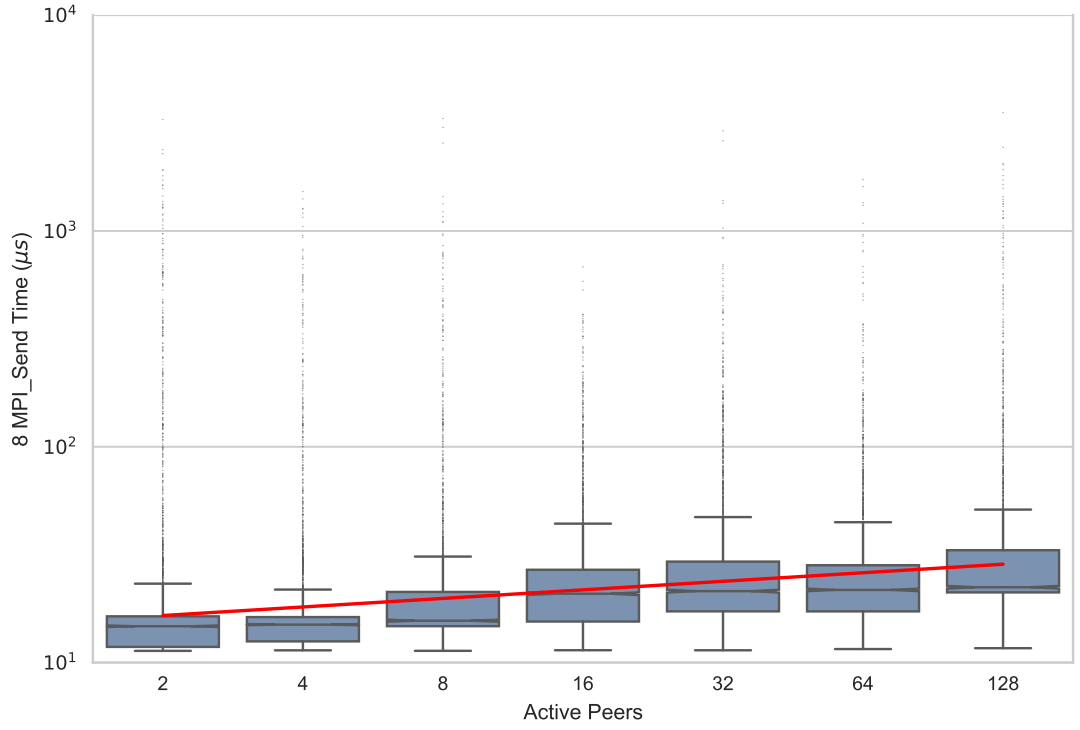


Figure 6.12: Distributions of insertion times using the *msgq* queue algorithm. The red line shows the linear regression through the 99 percentile data.

Chapter 7

Conclusion

This chapter will summarize the contributions within the thesis and presents further possible work.

7.1 Summary

This thesis investigated ways in which the latency and scalability of collective operations can be improved for modern supercomputers. These aspects of collective operations are extremely important. Many applications depend on small-message AllReduce operations, which are latency bound. In combination with the growing size of both supercomputers and the usage patterns, and the stagnation of improvements to interconnect latency this leads to a critical shortcoming in modern high performance computing. This work presents a novel performance model for small messages, introduces a generalisation of recursive doubling, *recursive multiplying*, and finally introduces a new queue mechanism for memory-limited environments.

The MPI Standard provides collective operations as a tool for application developers to describe the required computation and data movement without specifying the implementation. This allows the MPI library implementors to use many algorithms to fulfill the requirements. The recursive doubling algorithm is typically used for a small-message AllReduce which is bound to $\log_2 N$, and is widely considered the theoretical limit. However, we show that recursive doubling is based on a limited model which does not account for the features of a modern interconnect.

Interconnects have evolved to standalone entities which act as a significant component in the architecture, not simply a mechanism of transport. Congestion avoiding routing in hardware and low latency have enabled a class of algorithms which would be

disadvantageous on early generations, but perform better on modern networks. We introduce the pipelining latency-bandwidth model and the *recursive multiplying* method to exploit these advances and perform the top-level operations faster.

The pipelining latency-bandwidth model was introduced to model the *interface* to the underlying capabilities more explicitly in the model itself: this is a key difference to other models which attempt to capture the true underlying behaviour. The model captures the observation that many modern interconnects are able to send multiple small messages more efficiently than the traditional latency-bandwidth model can represent. This allows for a more straightforward derivation of algorithms which can exploit underlying hardware. In addition, we developed the Fennel simulator to explore the space of potential algorithms.

The *recursive multiplying* method, based on the pipelining latency-bandwidth model, shows that the $\log_2 N$ scalability is not the absolute limit and methods can be developed which circumvent the limitations of prior algorithms. The implementation presented in this work is based on the capabilities of the Cray XC30 system, ARCHER, and developed using the Cray DMAPP RDMA library. With *recursive multiplying* more memory is consumed, but it yields significant benefits for latency. For example, we outperform recursive doubling by 10% on the median for 128 processes. Recursive doubling is the algorithm used in MPICH, the most popular MPI implementation. The methodology of *recursive multiplying*, sending more messages per stage, is the primary underlying contribution which may yield many future algorithms.

The *recursive multiplying* method provides more flexibility compared to the recursive doubling method and when modelled using the pipelining latency-bandwidth model we show that an optimal value can be derived given the machine hardware parameters. The range of different schedules provided by the *recursive multiplying* method gives greater applicability to more situations whereas the recursive doubling method is limited to power of two and requires additional *fixing* algorithms to reduce the process count to the nearest power of two.

The single-consumer multiple-producer mechanism enables lower latency and higher scalability by reducing the impact of a shared remote receive queue between all processes. This is especially relevant to applications of MPI which are used in a memory-limited environment, such as embedded devices. MPI libraries present on supercomputers would only use the *scmp* method when little memory is available.

Going forward the algorithms introduced in this work are applicable to future networks, because we observe similar capabilities in other modern networks and expect

the trends of the performance characteristics in future interconnects to remain similar. With an improvement in the features which the introduced algorithms exploit we expect an even larger performance benefit.

7.2 Further Work

7.2.1 Performance Modelling

The pipelining latency-bandwidth model presented in Chapter 4 was able to represent the small-message behaviour very well for multi-casting, as used in Chapter 5. An extension to the model, hinted at in Section 4.3.2, would be to allow for α_r and α_p to be a function of message size. This would yield a better theoretical model on which to base work which includes varying sizes of messages. Various algorithms as described in Section 5.2 could thereby be modelled.

In addition to extending the model, the *Fennel* simulator could be extended to include more effects only seen in *real-world* machines. For example topology-based latency, or probability distributions for the latencies.

7.2.2 Recursive Multiplying

The *recursive multiplying* method was successful at implementing a small-message AllReduce. However, further improvements can be made to the algorithm. Currently the algorithm only supports fixed buffer sizes, but buffer splitting algorithms are common for large message sizes to achieve information distribution. By implementing notation similar to the current schedule notation to support buffer splitting, global AllReduce operations can be represented, acting on partial buffers. This, combined with the current schedules, would generalise to all currently known recursive doubling related algorithms. A clear approach to this would be determining schedules using a group theory approach similar to Kolmakov et al.[70].

The recursive doubling method is used in multiple implementations of MPI operations for small message sizes. Two simple extensions would be to use recursive multiplying for AllGather and ReduceScatter, with the caveat to only utilize it for appropriate message sizes. Another analogous extension is to implement the multi-way recursive halving method, which can be used as a building block for ReduceScatter.

The fundamental idea of recursive multiplying stems from the pipelining latency-bandwidth model, which allows us to send multiple messages cheaply. This idea can

be extended beyond the recursive multiplying implementation of it. Recursive multiplying is suited for barrier-type collective operations, within which all processes must communicate their information with all other processes. Sending multiple messages with redundant information can be extended to irregular collectives, or partitioned collectives[46].

7.2.3 Receive Queue Mechanism

The *SCMP* queue mechanism presented succeeds in reducing congestion for insertion into the receive queue, but emptying the receive queue is currently only done with a single thread. A multi-threaded approach to this would be ideal to convert the design into a remote-local multi-consumer multi-producer queue. Fortunately, this is done exclusively on the shared-memory of a single node, therefore a typical lock-less approach can be used to traverse the double buffered queue. The difficulty comes from synchronizing the switching between the remote and local processes.

Bibliography

[1] Adams, D. A. (1993). *CRAY T3D System Architecture Overview Manual*.

[2] Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. (1997). Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71 – 79.

[3] Alverson, B., Froese, E., Kaplan, L., and Roweth, D. (2012). Cray XC Series Network. Technical report, Cray Inc.

[4] Bagrodia, R., Deelman, E., and Phan, T. (2001). Parallel Simulation of Large-Scale Parallel Applications. *The International Journal of High Performance Computing Applications*, 15(1):3–12.

[5] Bar-Noy, A. and Kipnis, S. (1992). Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):431–452.

[6] Barnett, M., Littlefield, R., Payne, D., and van de Geijn, R. (1993). Global Combine on Mesh Architectures with Wormhole Routing. In *Proceedings Seventh International Parallel Processing Symposium*, pages 156–162.

[7] Barnett, M., Shuler, L., van de Geijn, R., Gupta, S., Payne, D. G., and Watts, J. (1994). Interprocessor collective communication library (InterCom). In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 357–364.

[8] Barrett, B. W., Brightwell, R., Grant, R. E., Hemmert, S., Pedretti, K., Wheeler, K., Underwood, K., Riesen, R., Maccabe, A. B., and Hudson, T. (2014). The Portals 4.1 Network Programming Interface. Technical Report April, Sandia National Laboratories.

[9] Beran, M. (1999). Decomposable Bulk Synchronous Parallel Computers. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics*, SOFSEM '99, pages 349–359, Berlin, Heidelberg. Springer-Verlag.

[10] Booth, S. (2001). Optimising the MPI Library for the T3E. In *Euro-Par 2001 Parallel Processing*, pages 80–83, Berlin, Heidelberg. Springer Berlin Heidelberg.

[11] Brooks, E. D. (1986). The butterfly barrier. *International Journal of Parallel Programming*, 15:295–307.

[12] Bruce, R., Chapple, S., MacDonald, N., Trew, A., and Trewin, S. (1995). CHIMP and PUL: Support for portable parallel computing. *Future Generation Computer Systems*, 11(2):211–219. Massive Parallel Computing.

[13] Buntinas, D., Mercier, G., and Gropp, W. (2006). Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, volume 1, pages 521–530.

[14] Calkin, R., Hempel, R., Hoppe, H.-C., and Wypior, P. (1994). Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615 – 632. Message Passing Interfaces.

[15] Cameron, K., Ge, R., and Sun, X.-H. (2007). LognP and log3P: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. *IEEE Transactions on Computers - TC*, 56:314–327.

[16] Cameron, K. L., Clarke, L., and Gordon, A. D. (1995). CRI/EPCC MPI for CRAY T3D. In *1st European Cray T3D Workshop*.

[17] Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K., and Livermore, L. (1999). Introduction to UPC and Language Specification. Technical report, Lawrence Livermore National Laboratory.

[18] Casanova, H., Giersch, A., Legrand, A., Quinson, M., and Suter, F. (2014). Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917.

[19] Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications*, 21(3):291–312.

[20] Chan, E., Heimlich, M., Purkayastha, A., and van de Geijn, R. (2007). Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783.

[21] Cook, S. A. and Reckhow, R. A. (1972). Time-Bounded Random Access Machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 73–80, New York, NY, USA. Association for Computing Machinery.

[22] Cray DMAPP (2012). *Cray XC Series GNI and DMAPP API User Guide*. Cray Inc. Accessed: 2020-09-10.

[23] Cray One (1977). *CRAY-1 COMPUTER SYSTEM*. Cray Research Inc.

[24] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: towards a realistic model of parallel computation. *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 28(7):1–12.

[25] Darema, F., George, D., Norton, V., and Pfister, G. (1988). A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1).

[26] De Sensi, D., Di Girolamo, S., McMahon, K. H., Roweth, D., and Hoefler, T. (2020). An in-depth analysis of the slingshot interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press.

[27] Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113.

[28] Dennard, R., Gaensslen, F., Yu, H.-N., Rideovt, V., Bassous, E., and LeBlanc, A. (2007). Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *Solid-State Circuits Newsletter, IEEE*, 12:38 – 50.

[29] Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.-C., Barkai, D., Berthou, J.-Y., Boku, T., Braunschweig, B., Cappello, F., Chapman, B., Xuebin Chi, Choudhary, A., Dosanjh, S., Dunning, T., Fiore, S., Geist, A., Gropp, B., Harrison, R., Hereld, M., Heroux, M., Hoisie, A., Hotta, K., Zhong Jin, Ishikawa, Y.,

Johnson, F., Kale, S., Kenway, R., Keyes, D., Kramer, B., Labarta, J., Lichnewsky, A., Lippert, T., Lucas, B., Maccabe, B., Matsuoka, S., Messina, P., Michielse, P., Mohr, B., Mueller, M. S., Nagel, W. E., Nakashima, H., Papka, M. E., Reed, D., Sato, M., Seidel, E., Shalf, J., Skinner, D., Snir, M., Sterling, T., Stevens, R., Streitz, F., Sugar, B., Sumimoto, S., Tang, W., Taylor, J., Thakur, R., Trefethen, A., Valero, M., van der Steen, A., Vetter, J., Williams, P., Wisniewski, R., and Yelick, K. (2011). The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60.

[30] Dongarra, J. and Luszczek, P. (2011). *TOP500*, pages 2055–2057. Springer US, Boston, MA.

[31] Ebcioglu, K. and et al. (2004). X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access.

[32] Edinburgh Parallel Computing Centre (2013). ARCHER. `https://www.archer.ac.uk`. Accessed: 2020-08-27.

[33] EMPI4Re (2017). `http://www.epigram-project.eu/empi4re/`. Accessed: 2017-04-05.

[34] End, V., Yahyapour, R., Simmendinger, C., and Alrutz, T. (2015). Adaption of the n-way Dissemination Algorithm for GASPI Split-Phase Allreduce. In *The Fifth International Conference on Advanced Communications and Computation*, pages 13–19.

[35] Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., and Reinhard, J. (2012). Cray Cascade: A scalable HPC system based on a Dragonfly network. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–9. IEEE Computer Society Press.

[36] Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

[37] Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). An Overview of the HDF5 Technology Suite and Its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, page 36–47, New York, NY, USA. Association for Computing Machinery.

[38] Fortune, S. and Wyllie, J. (1978). Parallelism in random access machines. *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118.

[39] Frigo, M. and Johnson, S. G. (2005). The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231.

[40] Gannon, P. (2006). *Colossus: Bletchley Park's Greatest Secret*. Atlantic Books.

[41] GASPI Forum (2013). *GASPI: Global Address Space Programming Interface*. Fraunhofer ITWM, 1 edition.

[42] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine*. The MIT Press.

[43] Geist, A. and Lucas, R. (2009). Major Computer Science Challenges At Exascale. *International Journal of High Performance Computing Applications*, 23(4):427–436.

[44] Gottlieb, A., Lubachevsky, B. D., and Rudolph, L. (1983). Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Trans. Program. Lang. Syst.*, 5(2):164–189.

[45] Graham, R. L., Woodall, T. S., and Squyres, J. M. (2006). Open MPI: A Flexible High Performance MPI. In Wyrzykowski, R., Dongarra, J., Meyer, N., and Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg. Springer Berlin Heidelberg.

[46] Grant, R. E., Dosanjh, M. G. F., Levenhagen, M. J., Brightwell, R., and Skjellum, A. (2019). Finepoints: Partitioned Multithreaded MPI Communication. In *High Performance Computing*.

[47] Gropp, W. (2002). MPICH2: A New Start for MPI Implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Berlin, Heidelberg. Springer-Verlag.

[48] Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R. D., Pritchard, H., and Squyres, J. M. (2015). A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 34–39.

[49] Hayes, J. P., Mudge, T., Stout, Q. F., Colley, S., and Palmer, J. (1986). A Microprocessor-based Hypercube Supercomputer. *IEEE Micro*.

[50] Hensgen, D., Finkel, R., and Manber, U. (1988). Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17.

[51] Hermanns, M.-A., Geimer, M., Wolf, F., and Wylie, B. (2009). Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 78 – 84.

[52] Hockney, R. (1977). Supercomputer architecture.

[53] Hockney, R. and Jesshope, C. (1981). *Parallel Computers*. Institute of Physics Publishing.

[54] Hockney, R. W. (1994). The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389 – 398.

[55] Hodges, A. (1983). *Alan Turing: The Enigma*. Burnett Books.

[56] Hoefler, T., Mehlan, T., Mietke, F., and Rehm, W. (2006a). Fast barrier synchronization for InfiniBand. *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.

[57] Hoefler, T., Mehlan, T., Mietke, F., and Rehm, W. (2006b). LogfP - a model for small Messages in InfiniBand. *Parallel and Distributed Processing Symposium*.

[58] Hoefler, T. and Rehm, W. (2005). A Communication Model for Small Messages with InfiniBand. In *PARS Mitteilungen*, pages 32–41.

[59] Hoefler, T., Schneider, T., and Lumsdaine, A. (2010). LogGOPSim: simulating large-scale applications in the LogGOPS model. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604.

[60] Hoefler, T., Siebert, C., and Lumsdaine, A. (2009). Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *International Conference on Parallel Processing*, ICPP '09, USA. IEEE Computer Society.

[61] InfiniBand Trade Association (2000). *InfiniBand architecture specification: release 1.0*.

[62] Ino, F., Fujimoto, N., and Hagihara, K. (2001). LogGPS: A parallel computational model for synchronization analysis. *ACM SIGPLAN Notices*, 36(7):133–142.

[63] Intel Inc (1985). *iPSC/1 User Guide*.

[64] International Business Machines (1956). *The Fortran Automatic Coding System for the IBM 704*.

[65] Jain, N., Leininger, M. L., Bhatele, A., Howell, L. H., Böhme, D., Karlin, I., León, E. A., Mubarak, M., Wolfe, N., and Gamblin, T. (2017). Predicting the performance impact of different fat-tree configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17*, pages 1–13.

[66] Kahn, D. (1967). *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner.

[67] Kale, L. V. and Krishnan, S. (1993). CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 91–108, New York, NY, USA. Association for Computing Machinery.

[68] Kielmann, T., Bal, H. E., and Verstoep, K. (2000). Fast Measurement of LogP Parameters for Message Passing Platforms. *Parallel and Distributed Processing*, 1800:1176–1183.

[69] Kim, J., Dally, W. J., Scott, S., and Abts, D. (2008). Technology-Driven, Highly-Scalable Dragonfly Topology. In *2008 International Symposium on Computer Architecture*.

[70] Kolmakov, D. and Zhang, X. (2020). A Generalization of the Allreduce Operation. arXiv, 2004.09362.

[71] Laguna, I., Marshall, R., Mohror, K., Ruefenacht, M., Skjellum, A., and Sultana, N. (2019). A Large-Scale Study of MPI Usage in Open-Source HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA. Association for Computing Machinery.

[72] Lambert, J. (1758). Observationes variae in mathesin puram. *Acta Helvetica Physico-Mathematico-Anatomico-Bota-nico-Medica*, 3.

[73] Lamport, L. (1977). Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806–811.

[74] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system.

[75] Lee, D. and Kalb, J. (2008). Network Topology Analysis. Technical report, Sandia National Laboratories.

[76] Leiserson, C. E. (1985). Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Trans. Comput.*

[77] Li, Z., Mills, P., and Reif, J. H. (1995). Models and resource metrics for parallel and distributed computation. *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, 2:51–60.

[78] Maggs, B. M., Matheson, L. R., and Tarjan, R. E. (1995). Models of Parallel Computation: A Survey and Synthesis. *Sciences-New York*.

[79] McCartney, S. (1999). *ENIAC: The Triumphs and Tragedies of the World's First Computer*. Walker & Co.

[80] Meiko (1989). In-Sun Computing Surface.

[81] Mellor-Crummey, J. (1987). Concurrent Queues: Practical Fetch-and-Phi Algorithms.

[82] Mitra, P., Payne, D., Shuler, L., van de Geijn, R., and Watts, J. (1995). Fast Collective Communication Libraries, Please. Technical report, University of Texas at Austin, USA.

[83] Moore, G. (1965). Cramming more components onto integrated circuits. *Electronics*, 38.

[84] Moritz, C. A. and Frank, M. I. (1998). LoGPC: Modeling Network Contention in Message-Passing Programs. *SIGMETRICS Perform. Eval. Rev.*, 26(1):254–263.

[85] MPI Forum (1993). *MPI: A Message Passing Interface*. MPI Forum.

[86] Mubarak, M., Carothers, C. D., Ross, R. B., and Carns, P. (2017). Enabling Parallel Simulation of Large-Scale HPC Network Systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100.

[87] Murphy, R. C., Rodrigues, A., Kogge, P., and Underwood, K. (2004). The structural simulation toolkit: A tool for bridging the ar chitectural/microarchitectural evaluation gap. Technical report, Sandia National Laboratories.

[88] National Aeronautics and Space Administration (2016). When computers were human. `https://www.nasa.gov/feature/jpl/when-computers-were-human`. Accessed: 2021-01-22.

[89] Numrich, R. W. and Reid, J. (1998). Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*.

[90] Poole, S. W., Hernandez, O., Kuehn, J. A., Shipman, G. M., Curtis, A., and Feind, K. (2011). OpenSHMEM - Toward a Unified RMA Model. In *Encyclopedia of Parallel Computing*, pages 1379–1391, Boston, MA. Springer US.

[91] Pritchard, H. and Gorodetsky, H. (2011). A uGNI-based MPICH2 nemesis network module for Cray XE computer systems. *Cray User Group Meeting, Fairbanks, Alaska*.

[92] Pritchard, H., Gorodetsky, I., and Buntinas, D. (2011). A uGNI-based MPICH2 nemesis network module for the Cray XE. *Recent Advances in the Message*, pages 110–119.

[93] Rabenseifner, R. (1997). A new optimized MPI reduce algorithm. `https://fs.hlrs.de/projects/par/mpi//myreduce.html`. Accessed: 2016-12-05.

[94] Rabenseifner, R. (1999). Automatic Profiling of MPI Applications with Hardware Performance Counters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 35–42, Berlin, Heidelberg. Springer Berlin Heidelberg.

[95] Rabenseifner, R. and Träff, J. L. (2004). More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 3241:36–46.

[96] Rodrigues, A., Murphy, R., Kogge, P., and Underwood, K. (2007). The structural simulation toolkit: A tool for exploring parallel architectures and applications. Technical report, Sandia National Laboratories.

[97] Rodriguez, G., Badia, R. M., and Labarta, J. (2004). Generation of Simple Analytical Models for Message Passing Applications. In Danelutto, M., Vanneschi, M., and Laforenza, D., editors, *Euro-Par 2004 Parallel Processing*, pages 183–188, Berlin, Heidelberg. Springer Berlin Heidelberg.

[98] Ruefenacht, M., Bull, M., and Booth, S. (2016). Generalisation of Recursive Doubling for AllReduce. In *Proceedings of the 23rd European MPI Users' Group Meeting on - EuroMPI 2016*, pages 23–31.

[99] Ruefenacht, M., Bull, M., and Booth, S. (2017). Generalisation of Recursive Doubling for AllReduce: Now with simulation. *Parallel Computing*, 69:24–44.

[100] Rüfenacht, M. (2015). Scalable two-sided mpi using rdma hardware. Master's thesis, University of Edinburgh.

[101] Rugina, R. and Schauser, K. E. (1998). Predicting the running times of parallel programs by simulation. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 654–660.

[102] Shamis, P., Venkata, M. G., Lopez, M. G., Baker, M. B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R. L., Liss, L., et al. (2015). UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE.

[103] Shroff, M. and van de Geijn, R. A. (2000). CollMark: MPI Collective Communication Benchmark. In *International Conference on Supercomputing*.

[104] Skjellum, A. and Leung, A. P. (1990). A Portable Multicomputer Communication Library atop the Reactive Kernel. *Proceedings of the Fifth Distributed Memory Computing Conference, 1990.*, 2:767–776.

[105] Sultana, N., Rüfenacht, M., Skjellum, A., Bangalore, P., Laguna, I., and Mohror, K. (2020). Understanding the use of message passing interface in exascale proxy applications. *Concurrency and Computation: Practice and Experience*.

[106] Swiss National Supercomputing Centre (2012). Piz Daint. `https://www.cscs.ch/computers/piz-daint/`. Accessed: 2020-08-27.

[107] Symons, A. and Narasimhan, V. L. (1995). Parsim-message passing computer simulator. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 2, pages 621–630 vol.2.

[108] Symons, A. and Narasimhan, V. L. (1997). Design and application of parsim — a message-passing computer simulator. *IEEE Proceedings - Computers and Digital Techniques*, 144:7–14(7).

[109] Thakur, R. and Gropp, W. (2003). Improving the performance of collective operations in MPICH. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*.

[110] Thakur, R., Rabenseifner, R., and Gropp, W. (2005). Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66.

[111] Thomas Haigh, Mark Priestley, C. R. (2016). *ENIAC in Action: Making and Remaking the Modern Computer*. The MIT Press.

[112] Tikir, M. M., Laurenzano, M. A., Carrington, L., and Snavely, A. (2009). PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In Sips, H., Epema, D., and Lin, H.-X., editors, *Euro-Par 2009 Parallel Processing*, pages 135–148, Berlin, Heidelberg. Springer Berlin Heidelberg.

[113] Valiant, L. (2008). A Bridging Model for Multi-core Computing. *Journal of Computer and System Sciences*, 77:154–166.

[114] Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

[115] Valois, J. D. (1994). Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69.

[116] van Duijn, M., Visscher, K., and Visscher, P. (2016). BSPLib: a fast, and easy to use C++ implementation of the Bulk Synchronous Parallel (BSP) threading model. `http://bsplib.eu/`.

[117] Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

[118] von Neumann, J. (1945). First Draft of a Report on the EDVAC. Technical report, Los Alamos National Laboratory.

[119] Welchman, G. (1984). *The Hut Six Story*. Penguin.

[120] Zhang, Y., Chen, G., Sun, G., and Miao, Q. (2007). Models of parallel computation: A survey and classification. *Frontiers of Computer Science in China*, 1:156–165.

[121] Zheng, G., Gunavardhan Kakulapati, and Kale, L. V. (2004). BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*